National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

*Your file Votre référence*

*Our file Notre référence*

## NOTICE

**The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.**

**If pages are missing, contact the university which granted the degree.**

**Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.**

**Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.**

## AVIS

**La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.**

**S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.**

**La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.**

**La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.**

Canada

University of Alberta

# A Multiprocessor Based Controller for the PUMA 560 Robot

by

William Darin Ingimarson

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

Department of Electrical Engineering
Edmonton, Alberta
Spring 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

*Your file Votre référence*

*Our file Notre référence*

**The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.**

**L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.**

**The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.**

**L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.**

ISBN 0-315-82252-X

Canada

UNIVERSITY OF ALBERTA

*RELEASE FORM*

NAME OF AUTHOR: William Darin Ingimarson
TITLE OF THESIS: A Multiprocessor Based Controller for the PUMA 560 Robot

DEGREE: Masters of Science
YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication rights and other rights in association with the copyright of the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) Darin Ingimarson

Permanent Address:
5119 39th Street,
Innisfail, Alberta,
Canada

Date: April 22, 1993

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Multiprocessor Based Controller for the PUMA 560 Robot** submitted by **William Darin Ingimarson** in partial fulfillment of the requirements for the degree of Masters of Science.

V. G. Gourishankar

Dr. V. G. Gourishankar (Supervisor)

R. E. Rink

Dr. R. E. Rink

Hong Zhang

Dr. H. Zhang

Date: April 21 1993

# Abstract

The design, fabrication, and testing of a multiprocessor based controller for the Unimate PUMA 560 robot manipulator is the main topic of this thesis. The controller is designed to provide a flexible and reconfigurable platform for implementing sophisticated robot control algorithms. The controller handles convenient selection of user defined control algorithms via a multi-window user interface implemented under the X Window System. The controller also supports a control variable logging facility and graphical plotting of controller response to aid in control algorithm analysis.

This thesis also reviews current multiprocessor robot controllers. A description of the real-time operating system selected for the project is given, and the hardware and software designs for the project are described. Experimental results for a test scenario of control algorithm evaluation are included.

# Acknowledgements

I would like to express my gratitude towards my supervisor, Dr. V. G. Gourishankar, for his continued guidance and encouragement (not to mention patience) throughout the duration of this project. In addition to Dr. Gourishankar, I would also like to thank Dr. R. E. Rink and Dr. H. Zhang for agreeing to sit on my defence committee.

The many hours of technical assistance and advice received from David Stewart of Carnegie Mellon University were greatly valued and appreciated. Without his help, this project could not have achieved what it has.

In addition to the above, there are a number of other people to whom I would like to offer my thanks. For the marathon coffee sessions and "burger and beer" stress therapy I would like to thank my fellow graduate students Don Murray and Gloria Chow. I would also like to express my appreciation and thanks to Mitch Peacock for the "fishing trips" and for being a friend that I could always depend on. Mere thanks could never repay the support, patience, and understanding Carolyn Walker has shown during the time I have spent on this project. She has made many sacrifices and spent many weekends and nights alone without complaint or criticism while I was burning the midnight oil. And last, but certainly not least, I would like to thank my parents Barbara and Bill for their support and encouragement during this often stressful time.

Without my "support group" of friends and family I doubt that this thesis would have been completed. I will be forever indebted to these people for their support.

The financial assistance for this project cannot go without mention, as the project would not have been possible without it. The Department of Electrical Engineering, University of Alberta provided financial support via teaching assistantships and equipment grants, and the Natural Sciences and Engineering Research Council of Canada (NSERC) provided support through operating and equipment grants to my supervisor.

# Contents


**1 Introduction** **1**

1.1 Kruszewski's Controller . . . . . 6

1.2 Limitations of Kruszewski's Controller . . . . . 8

1.3 Proposed Controller . . . . . 9

1.4 Thesis Outline . . . . . 11

**2 Literature Survey** **13**

2.1 Introduction . . . . . 13

2.2 Techniques and Concepts Used in Multiprocessor Based Robot Controllers . . . . . 14

2.3 Overview of Coarse-Grained Multiprocessor Based Robot Controllers . . . . . 19

2.3.1 Heterogeneous Controllers . . . . . 19

2.3.2 Homogeneous Controllers . . . . . 22

2.3.3 Concluding Remarks . . . . . 29

**3 Design Methodology** **32**

3.1 System Requirements . . . . . 32

3.2 Design Method . . . . . 33

3.3 Data Flow Analysis . . . . . 34

3.4 Task Structuring . . . . . 35
3.5 Mapping of Modules to Hardware . . . . . 41
3.6 The Real Time Operating System . . . . . 42
3.7 Local Configuration of Chimera II . . . . . 46
3.8 Assignment of Task Priorities . . . . . 47

**4 Controller Hardware** **50**
4.1 Introduction . . . . . 50
4.2 Design Issues . . . . . 51
4.3 Non-Real-Time System . . . . . 51
4.4 Real-Time System . . . . . 54

**5 Controller Software** **56**
5.1 Design Goals . . . . . 56
5.2 Software Module Description . . . . . 58
5.2.1 User Interface . . . . . 58
5.2.2 Robot Manager . . . . . 59
5.2.3 Motion Generator Task . . . . . 60
5.2.4 Controller Task . . . . . 66
5.2.5 Data Logger Task . . . . . 67
5.2.6 Prefilter and Hardware Status Task . . . . . 68
5.3 Configuration Files . . . . . 71
5.3.1 Controller Configuration File . . . . . 71
5.3.2 State Variable Table Configuration File . . . . . 72
5.3.3 DVME-601 Configuration File . . . . . 73
5.3.4 Home Position File . . . . . 74
5.4 RTS to NRTS Communication Library . . . . . 74
5.5 Device Libraries . . . . . 78

5.6 Device Drivers . . . . . 79
5.6.1 Device Driver for the DVME-601 - dadc.c . . . . . 81
5.6.2 Device Driver for the DVME-628 - ddac.c . . . . . 82
5.6.3 Device Driver for the VMEbus Encoder Board - veb.c . 82
5.7 Description of Executable Files . . . . . 84

**6 Results** **85**
6.1 Introduction . . . . . 85
6.2 Timing and Testing of Libraries and Device Drivers . . . . . 85
6.3 Implementation of the PID Control Algorithm . . . . . 87
6.3.1 Tuning the PID Controller . . . . . 90
6.4 Tracking Performance of PID Control Algorithm . . . . . 92

**7 Conclusions and Suggestions for Future Research** **100**
7.1 Features of the MRC . . . . . 100
7.2 Suggestions for Future Research . . . . . 101

**Bibliography** **104**

**A System Configuration** **113**
A.1 System Memory Maps . . . . . 113
A.2 Board Configurations . . . . . 115
A.2.1 Bit-3 Model 412 VMEbus to VMEbus Adapter . . . . 115
A.2.2 Ironics IV-3220 Single Board Computer . . . . . 115
A.2.3 Datel DVME-601 Analog to Digital Converter . . . . . 118
A.2.4 Datel DVME-628 Digital to Analog Converter . . . . . 120
A.2.5 VMEbus Encoder Board . . . . . 120
A.3 CHIMERA II Configuration File . . . . . 122

**B PID Controller Module and Configuration Files** **125**

B.1 Servo Server Module for PID Controller . . . . . 126

B.1.1 Module Source . . . . . 126

B.2 Configuration files for Robot Controller . . . . . 137

B.2.1 Controller Configuration File . . . . . 137

B.2.2 State Variable Table Configuration File for PID Controller 138

B.2.3 DVME-601 Configuration File . . . . . 139

B.2.4 Home Position File . . . . . 139

# List of Tables


2.1 Heterogeneous multiprocessor robot controllers. . . . . 20

2.2 Homogeneous multiprocessor robot controllers . . . . . 31

6.3 Execution times for time-critical functions. . . . . 86

6.4 Gains for the PID controller . . . . . 91

A.5 System Memory map for A32 address space. . . . . 114

A.6 System Memory map for A24 address space. . . . . 116

A.7 System Memory map for A16 address space. . . . . 117

A.8 Local jumper settings for the Bit3 VMEbus to VMEbus Adapter . 117

A.9 Base Address Settings for the IV-3220 SBCs . . . . . 117

A.10 Jumper settings for the DVME-601 A/D Converter . . . . . 118

A.11 Port Assignments for the DVME-601 A/D Inputs . . . . . 119

A.12 Jumper settings for the DVME-628 D/A Converter . . . . . 120

A.13 Port Assignments for the DVME-628 D/A Outputs . . . . . 121

A.14 Jumper settings for the VMEbus Encoder Board . . . . . 121

# List of Figures


1 The PUMA 560 Industrial Robot, and peripherals. . . . . 4

2 Block diagram of the Unimation Control Computer. . . . . 5

3 Kruszewski's Controller for the PUMA 560. . . . . 7

4 The top-level DFD for the MRC. . . . . 36

5 Communication and Synchronization Symbols. . . . . 39

6 Task Diagram for the MRC. . . . . 40

7 Hardware for the Multiprocessor Based Robot Controller . . . . . 52

8 Frequency Response of the Prefilter . . . . . 69

9 Joint 1 Motion and Control Signal at 50% speed . . . . . 94

10 Joint 2 Motion and Control Signal at 50% speed . . . . . 95

11 Joint 3 Motion and Control Signal at 50% speed . . . . . 96

12 Joint 4 Motion and Control Signal at 50% speed . . . . . 97

13 Joint 5 Motion and Control Signal at 50% speed . . . . . 98

14 Joint 6 Motion and Control Signal at 50% speed . . . . . 99

15 Modified VMEbus Encoder Board jumper layout. . . . . 123

16 Modified VMEbus Encoder Board address decoding wiring. . . . 124

# Chapter 1

# Introduction

In order to pursue research in robot manipulator control, it is necessary to have a suitable hardware and software environment to test various control algorithms. Some control algorithms may require only simple position feedback, while others may require more complicated types of feedback such as joint velocity, acceleration or torque. Vision-oriented controllers will of course require the input from an image processing system, and computationally intensive control algorithms may require additional processing capability. These criteria imply that a controller that is to be used in a research environment be easily expandable, adaptable, and capable of being reconfigured. Unfortunately, most industrial robots come from the factory with a simple, built in control law such as individual joint PID control [Cra89], and lack the required flexibility for research purposes [MB89, GC88]. The Robotics Control Group of the Department of Electrical Engineering purchased a Unimation PUMA 560 Industrial Robot in 1988 for educational and research purposes. However, the Unimation controller suffers from the aforementioned limitations of typical industrial robot controllers.

The PUMA 560 is a six degree of freedom revolute joint industrial robot manipulator. The complete system as supplied by the manufacturer consists of the robot manipulator, a rack-mountable control computer, a teach pendant, peripherals (such as I/O modules or a diskette drive), and software required to control the robot [Uni85]. A terminal for issuing commands and editing robot control programs in the VAL-II operating system may be connected to the control computer via a 9600 baud serial port. Communication with external on/off devices and sensors is handled by the I/O module. A diskette drive is used to store robot control programs and program data. The teach pendant is used to manually move the robot through a task and record those motions in the control computer for later playback. The PUMA 560 robot manipulator and its peripherals are shown in Figure 1.

The control computer is composed of three main sections. A DEC LSI-11 computer, six digital servo control boards (one for each joint of the arm), and the six power amplifiers (three on each power amplifier assembly). The LSI-11, which resides on a standard DEC LSI-11 Bus backplane along with its memory and I/O cards, performs system supervision and path planning tasks. An "A" Interface card connects the LSI-11 Bus to the proprietary Unimation Servo Bus, and to the six digital servo cards located there, via a "B" Interface card residing on the latter bus. See Figure 2 for the block diagram of the Unimation control system.

Each servo card contains a hardcoded PID control algorithm running on a Rockwell 6503 microprocessor [Cra89]. The PID controllers use the angular position of the robot joints as their feedback signal. The reference angle is supplied by the LSI-11, and the control outputs (in the form of voltages) are sent to the joint amplifiers. The outputs of the amplifiers drive the manipulator's six permanent magnet DC servomotors. It is not possible to easily

modify either the path-planning software running on the DEC LSI-11 or the control algorithms on the 6503 servo cards.

Each servomotor possesses two types of positional sensor; an optical encoder which generates a quadrature signal used to drive an up-down counter, and a potentiometer [Uni85]. Each motor's optical encoder provides a high-resolution position signal to its respective joint controller card by means of an up-down counter actuated by the quadrature signal. The potentiometers are used to determine absolute joint position of the robot arm in order to initialize the up-down counters. The three major joints of the robot arm, the 'waist', 'shoulder' and 'elbow', possess electromagnetic brakes which support the arm when power is removed from the servomotors.

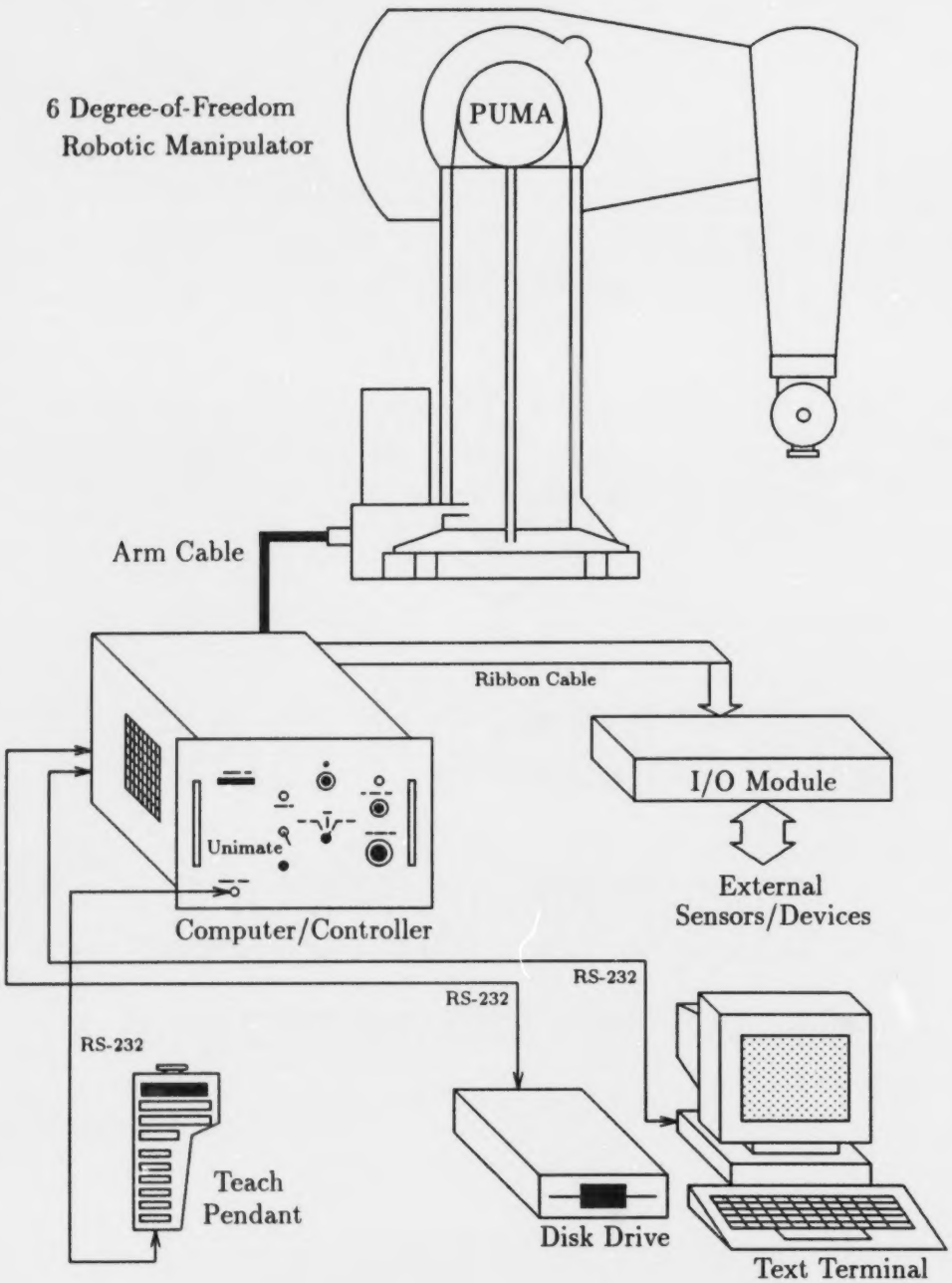


Figure 1: The PUMA 560 Industrial Robot, and peripherals.

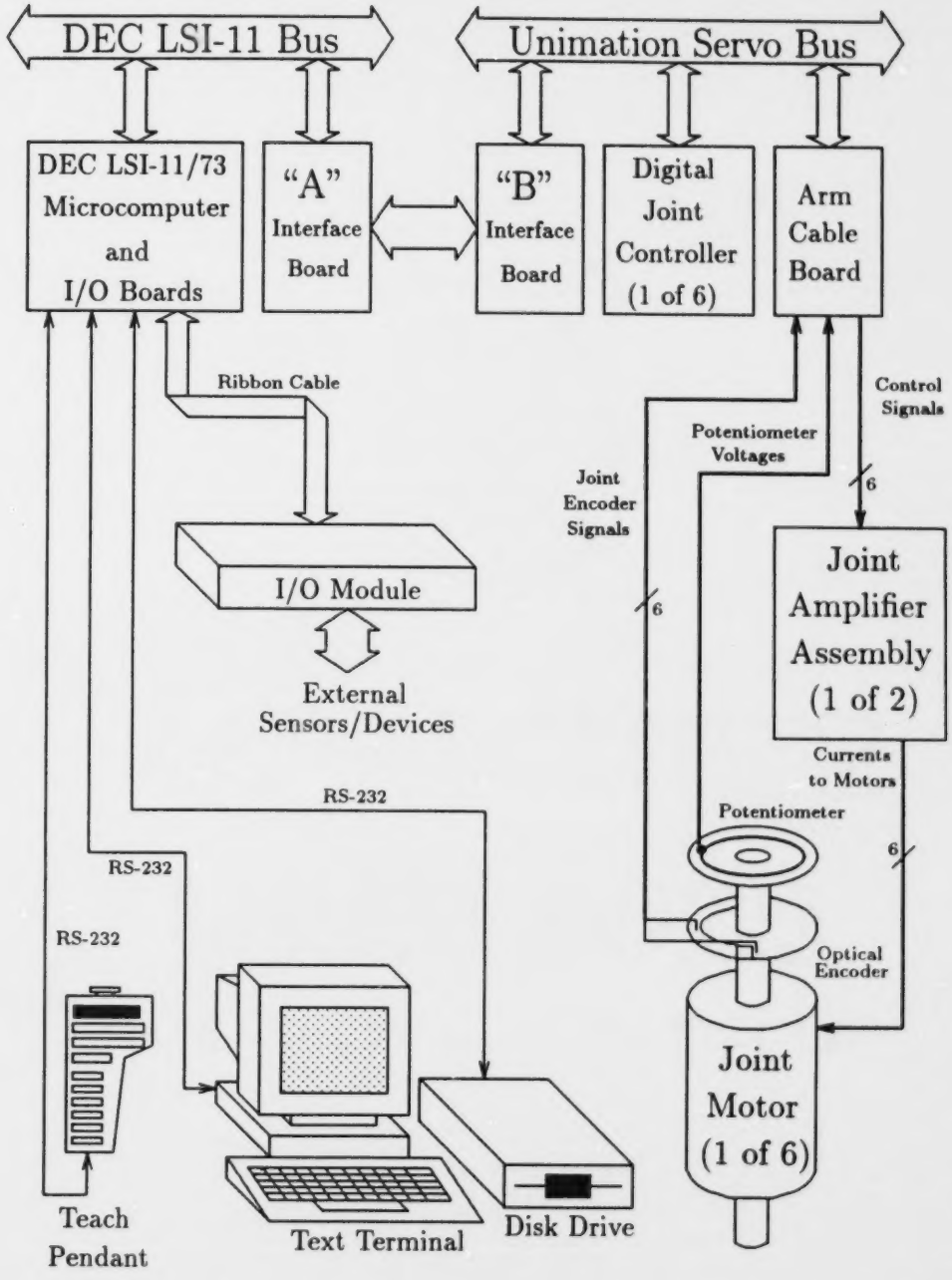


Figure 2: Block diagram of the Unimation Control Computer.

## 1.1 Kruszewski's Controller

A previous research project expanded on the capabilities of the PUMA equipment purchased by the Robotics Control Group [Kru90]. The the objective of that project was to replace the Unimation controllers and the DEC LSI-11 with a controller implemented on a 68000 based CPU resident in an external VMEbus cardcage. For convenience this controller will be referred to as Kruszewski's Controller.

The main components of this controller are depicted in Figure 3. The TUTOR CPU card is a Motorola MC68000 based single board computer for the VMEbus developed by the University of Alberta's Department of Electrical Engineering. This card performed all path planning and control functions using a timer-driven interrupt scheme. The RIOT card (the companion RAM, I/O and timer card to the TUTOR) was used to provide access to the SUN 3/160 development platform for uploading and downloading software. It also provided user interaction with the controller and the Datel DVME-601 A/D Board via a text-only dumb terminal. The AMLine Decoder Board was necessary to convert non-standard address modifier codes generated by the TUTOR board into codes that the other VMEbus boards could understand.

The Datel DVME-601 A/D Converter sampled the joint potentiometer voltages and the hardware status-line voltages provided by the PIB. The DVME-601 ran a simple four-point batch-averaging prefilter on the pot voltage signals, and then presented these values along with the status (high or low) of the three sampled PUMA hardware status lines for use by the TUTOR CPU card. The Datel DVME-628 D/A Converter supplied joint control voltages and an arm power enable signal to the PIB which in turn buffered these signals and applied them to the proper points in the modified Unimation

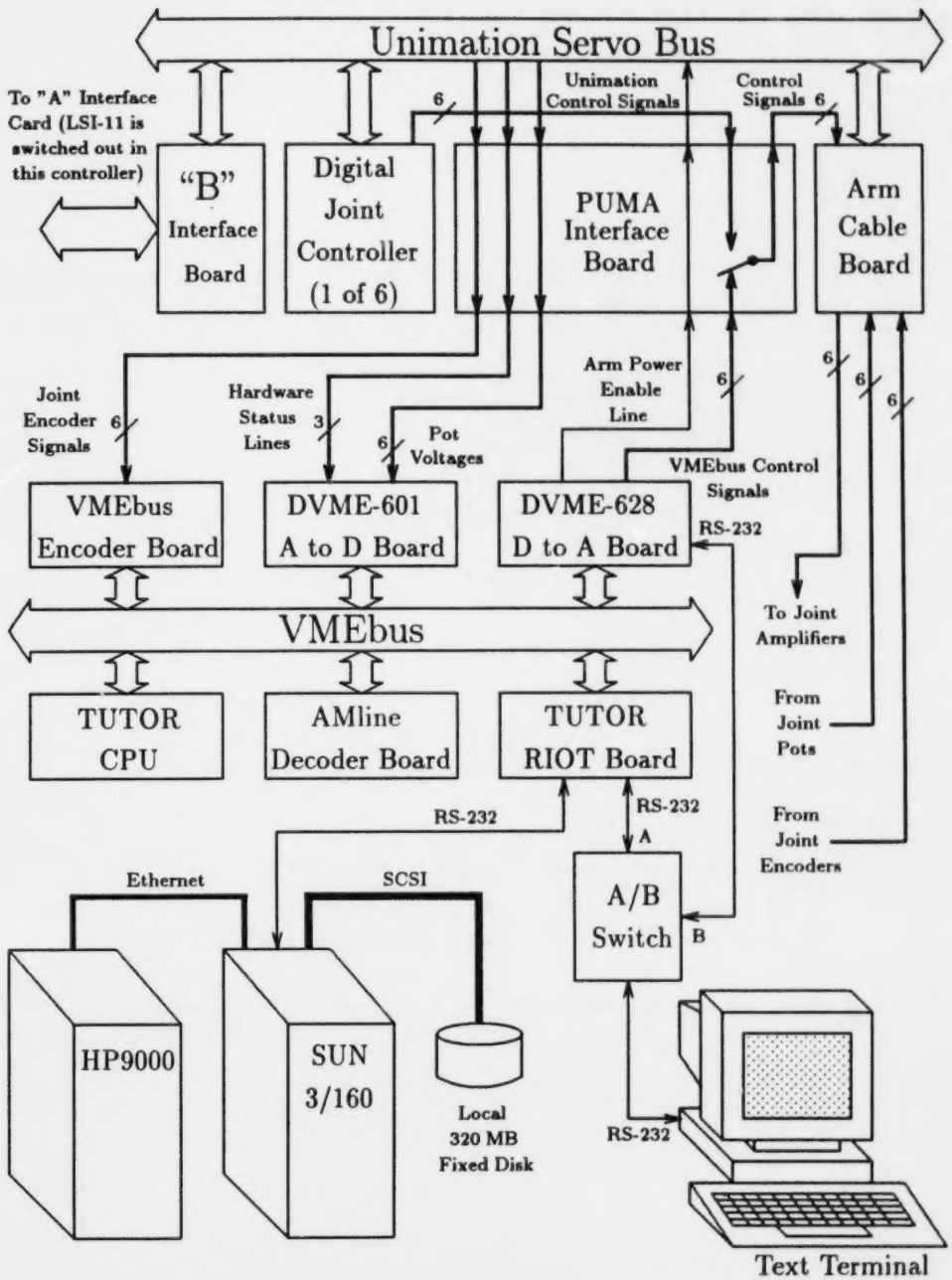


Figure 3: Kruszewski's Controller for the PUMA 560.

Control Computer chassis. The VMEbus Encoder Board used the joint encoder quadrature signals supplied to it by the PIB to drive up-down counters containing integer counts representing the robot's joint angles.

Programs for the controller were compiled on the Department of Electrical Engineering's HP9000 computer and then downloaded via Ethernet to the local SUN 3/160. The programs were then downloaded to the VMEbus system over 9600 baud serial lines. For more detailed information on this project, please refer to [Kru90].

## 1.2 Limitations of Kruszewski's Controller

Kruszewski's Controller had several limitations which made its use as a test platform for sophisticated control algorithms difficult at best. While adequate in processing ability for very simple control algorithms (for example, an integer arithmetic PD controller), the TUTOR CPU was not capable of providing the computing power for more complex algorithms. In order to support research into more advanced controller algorithms, it is necessary to increase the processing power of the controller, and to implement software able to take advantage of the increased computing capabilities.

Due to the limitations in processing power available in Kruszewski's Controller, the use of floating-point calculations in conjunction with fast sampling rates in the controller algorithm was not possible. Because only integer gains could be used in tuning operations [Kru90], the controller could not be tuned using the widest possible control gain range.

Also, Kruszewski's Controller could not use the Robotics Lab's local workstations for compiling control programs. This meant that the Department of Electrical Engineering's HP9000 computer had to be used for all compiling

tasks. The compiled code then had to be downloaded to the local Sun 3/160 workstation via an Ethernet link, and then through a serial interface to the VMEbus system.

Modification of the control software to support different control algorithms involved repeating the complicated compilation and download procedure each time a different control algorithm was to be implemented and tested. The source code for the controller was not written in a consistent, modular fashion, and consisted of one large, unwieldy file. The declarations of the controller variables themselves were in a part of this monolithic source quite far removed from the control code itself, making reading and modification of the source code difficult.

Kruszewski's Controller contained no provision for making use of the potential of the SUN 3/160 workstation as a graphical front end to the research system. All user interface operations were performed via a text terminal that lacked the ability to display plots of logged data, or provide windowed displays of multiprocessor operations and status.

## 1.3 Proposed Controller

This project addresses the lack of processing power in Kruszewski's Controller by replacing the TUTOR CPU card with two Ironics IV-3220 MC68020 based single board computers (SBCs), each with an MC68882 Floating Point Coprocessor. These two SBCs, each one possessing more computing power than the TUTOR CPU, are operated in parallel to provide the processing capabilities required by sophisticated control algorithms. The floating point coprocessors speed the floating point calculations used in controller modules, allowing much higher sampling rates than possible with the TUTOR CPU.

The VMEbus cardcage is linked to the Sun 3/160 workstation by means of a Bit3 Model 412 VMEbus to VMEbus Adapter. This allows direct high-speed communication between the host workstation and the real-time VMEbus system. The compiler on the workstation is used to compile real-time programs for the VMEbus system. The compiled code is then downloaded, via the Bit3 link, to the real-time system where it executes. The adapter provides a more convenient path for interprocessor communication between the host and the real-time processors than the serial link of Kruszewski's Controller. This link also allows the real-time system to use of many of the host workstation's features, a goal not possible with the much slower serial communication used in Kruszewski's Controller. The X Window System, which runs on the host workstation, is used as a multi-window interface to the multiprocessor controller. The graphics capabilities of the workstation are also used to display plots of logged data from the real-time system.

The simple interrupt driven kernel in Kruszewski's Controller has been replaced with the Chimera II Real-Time Operating System from Carnegie Mellon University. Chimera II provides software libraries and utilities to support the hardware with a real-time programming environment. Standard real-time operating system (RTOS) features such as semaphores, lightweight tasks, interprocessor communication, and advanced process scheduling are available. Chimera II also provides features specially designed for sophisticated robot controllers such as specialized state variable shared memory segments, task management functions for dynamic periodic tasks, and a matrix manipulation library. Chimera II also utilizes device drivers written in the C programming language, and library support for reconfigurable real-time systems; valuable features for a research oriented controller.

All programs are written in the C programming language and extensive use

of object oriented programming techniques is made to improve the programming interface for use by robot control researchers. The use of the Chimera II operating system enables the bulk of the development operations to be moved from the HP9000 to the local Robotics Lab workstations.

The new controller, hereafter referred to as the Multiprocessor Robot Controller (MRC), will allow more sophisticated control algorithms to be operated at much higher sampling rates than was possible with Kruszewski's Controller. The MRC is easily able to run a floating-point PID controller at 606 Hz, and is able to control all six PUMA joints. Kruszewski's Controller implemented an integer PD control algorithm for the first three joints of the arm, and ran at 333 Hz.

## 1.4 Thesis Outline

In Chapter 2 a literature review of previous research efforts in multiprocessor robot controllers is given. Chapter 3 describes in detail the overall system design of the controller. Included in this chapter are the chosen design method, the statement of system requirements, the data flow diagram, and the task diagram. Chapter 3 also gives a brief overview of Chimera II, the real-time operating system chosen to support this project. Chapter 4 describes the configurations and modifications made to the system hardware. Chapter 5 gives a detailed description of the software solution implemented to satisfy the system requirements. Chapter 6 presents the results of timing the controller libraries as well as a test scenario using the MRC as a test platform. Preliminary tuning of a PID control algorithm is presented, and the behavior of the controller while servoing the manipulator joints is examined. Conclusions and suggestions for further extensions to the system are given in Chapter 7.

Detailed hardware configuration information may be found in Appendix A. The source code for the PID module used to obtain the results as well as its associated software configuration files can be found in Appendix B.

Volume 2 of this work contains information specific to the software component of the project and is available from the Department of Electrical Engineering's Robotics Lab. It contains a detailed description of the operation of Chimera II, the Software User's Guide for the controller software and various utility programs, and descriptions of the programming interface provided by the second-level device drivers. A complete source listing of the MRC software is also included in Volume 2.

# Chapter 2

# Literature Survey

## 2.1 Introduction

Modern robotics grew from the technologies of telecherics and numerically controlled machines [GWNO86]. Telecherics, the field of using remote manipulators [1], is used extensively in the handling of hazardous materials. The first telecheric devices were solely mechanical, but modern remote manipulators often combine mechanical systems with electronic feedback control [GWNO86]. Numerically controlled (NC) machines utilize numeric input to control their operations. Early NC machines were programmed with adjustable stops or cams, punched tape or cards, or magnetic media.

A robot is a mechanical manipulator that is operated by numerical control methods. The first robotic devices utilized mechanical cams and limit switches for control. Mainframe and minicomputer technology was later applied to the control of robots, but only in research environments [GWNO86]. The availability of relatively inexpensive microprocessors in the mid 1970's allowed

[1] Remote manipulators are often referred to as teleoperators.

economical implementation of parallel microprocessor control architectures. Current research in robot controllers includes fields such as multiprocessor based control systems, parallel programming techniques for robot controllers, and reconfigurable real-time systems.

This chapter begins with an overview of some of the techniques and concepts that are important in multiprocessor based robot control systems. Tabular comparisons of various efforts in multiprocessor robot controllers are presented for cases demonstrating coarse-grained task division among processors. A brief discussion of components and techniques used in realizing each of these systems balances the tabular comparison. Conclusions on current research directions in the field of coarse-grained multiprocessor robot controllers are drawn from the surveyed literature.

## 2.2 Techniques and Concepts Used in Multiprocessor Based Robot Controllers

The overall performance of a digital robot controller, in terms of system stiffness, is to a large extent dependent upon its sampling rate [Kho87]. However, in a research environment, the control algorithm calculations can become very computationally intensive when more sophisticated control algorithms are implemented. This correspondingly decreases the maximum achievable sampling rate, and with it, system performance decreases. A single processor controller will quickly reach the limits of its ability and require either a reduced sampling period, use of integer arithmetic [Kru90], or an alternate solution in terms of a computing platform. One method of providing the requisite computing power is through the use of multiprocessing architectures [WKKT86, LS88, Gra89].

This type of computing architecture more closely matches the asynchronous nature of real world problems such as the control of a robot manipulator [GWM+].

The use of multiple processors implies the need to divide the labor of controlling the robot among the available CPUs. The division of labor can take advantage of the parallelism inherent in a robot at different levels. The task division may occur at the algorithmic level (fine granularity) using pipelined architectures or systolic processor arrays [LS88, Gra89, Lat85], or the task division may occur at a coarser level of granularity and use general purpose microprocessors as computing engines.

In a coarse-grained system, the computational task is divided into a number of processes which are allocated to the various CPUs in the system. One example of such a system, presented by Paul and Zhang in [PZ86], uses general purpose microprocessors to provide force and motion control for a robot manipulator. Another example of this type of architecture is the Chimera II system [SKHK89, SSK89, SSK90, SSK92] which uses general purpose microprocessors supported by a real-time kernel to implement a multiprocessor/multitasking real-time system.

In order to effectively decompose the task of controlling a robot into processes, a model such as the NASA/NBS Standard Reference Model for Telerobotic Control System Architecture (NASREM) may be used. The model presents a standard, hierarchal set of control levels with well-defined interfaces [MW90]. Each level is implemented as a set of three, distinct modules; sensory processing, world modeling, and task decomposition [SSK90]. The sensory processing module obtains and integrates information about the system (i.e. reading data from sensors). The world modeling module organizes and controls access to a global database which provides information needed by

the control level. The task decomposition module performs a particular function (depending on the control level) based on data provided by the sensory processing module and the world modeling module. Six such control levels are defined for the NASREM standard, from coordinate transform and servo level tasks to mission planning tasks. Generally, the NASREM standard is simplified for actual implementations by removing some of the higher control levels that are not required [MB89].

Other real-time software design methods exist that may be applied to robot controllers. One such method is the DARTS (Design Approach for Real-Time Systems) method [Gom84] which applies transform grouping criteria to a traditional Data Flow Diagram to produce a set of concurrent tasks with minimized intertask coupling. This method of design was used to realize the controller developed for this project. A more detailed description of this method as applied to this project is provided in Chapter 3.

Coarse grained multiprocessor systems may be classified into two groups: heterogeneous systems and homogeneous systems. Heterogeneous systems contain processing elements of different types, such as DSPs or CORDIC (trigonometric) processors. In this type of system, each processor type is typically geared to a specific job, such as calculation of inverse dynamics, or generation of joint control voltages. While the heterogeneous systems tend to exhibit higher performance in terms of throughput, they often suffer from a clumsy user interface and require special purpose compilers and interface code [SKHK89], not to mention special hardware support [MWB89].

Homogeneous systems are composed of processors of one type, or at least, of one family. Homogeneous systems provide a more convenient platform for implementing interprocessor communication and they require fewer support resources, in terms of specialized interface hardware, compilers, and inter-

face code. The Chimera II system can be classed as a homogeneous system. Although work is proceeding on supporting heterogeneous processor configurations with this system, the real-time operating system kernel runs on only one family of microprocessor [SSK90].

A multiprocessor system, be it heterogeneous or homogeneous, requires some means of interprocessor communication and synchronization. The choice of communication methods usually lies between shared memory or message passing. Message passing is the more restrictive method, and also requires the added overhead of communications protocol [LS88]. Shared memory, since its speed may approach the maximum speed of the bus for small numbers of processors, is the preferred method [LS88]. Semaphores are used as synchronization mechanisms, or to pass signals between processes. Semaphores are also used to provide mutual exclusion in cases where more than one process is in contention for a resource.

Interprocess and interprocessor communication, semaphores, and task scheduling (for instances where multiple real-time tasks may exist on a single CPU) may be realized in several ways. Some microprocessors, such as the INMOS T800, provide hardware task schedulers and specialized interprocessor communication mechanisms especially designed for multiprocessor architectures. Some systems realize the required facilities via a runtime software library of precompiled functions, such as is the case with the NYMPH system [CFAB86]. The runtime library avoids the overhead of an actual operating system while still implementing most of the required functionality.

Other multiprocessor systems, such as Chimera II and CONDOR [NSH88, NSH89] utilize a real-time operating system to provide certain features required by the user. These systems support task manipulation, interprocess and interprocessor communication, and hardware independence through the

use of device drivers and libraries. Standard compilers are used to develop real-time programs, and the user interfaces are reasonably convenient to use. This provides a sophisticated environment in which to implement real-time robot control tasks.

The convenience of the real-time multiprocessor operating system does not come without a price tag however. The operating system requires a certain amount of CPU time in which to execute its context switches and scheduling algorithms. High-level languages, while often nearly as fast as assembly code, do not produce programs which run as fast as hand-coded ones. The use of a real-time operating system in a multiprocessing environment usually precludes a heterogeneous system. The literature surveyed did not indicate any RTOS that allowed its kernel to be run on multiple processors in a heterogeneous system. The usual method of supporting special purpose processors under a multiprocessor RTOS is to treat the special purpose processors as slaves to the CPUs running the real-time kernel.

Certain multiprocessor configurations do not require the services of a real-time operating system. For example, the multiprocessor system for calculation of real-time inverse-dynamics described in [WCLL89] is coded completely in floating-point assembly code with each subtask program coded as a macro. Each subtask executes at a certain predetermined time and since the subtasks are arranged to execute serially on each of the four CPUs, no task scheduling is necessary, and no real-time operating system is required.

The ability to statically or dynamically reconfigure a real-time system can be extremely useful in a research environment. The study of reconfigurable real-time systems [SVK92] deals in part with this ability. These types of systems tend to be quite abstracted from their hardware platforms, and make extensive use of object-oriented programming techniques.

## 2.3 Overview of Coarse-Grained Multiprocessor Based Robot Controllers

This section provides a tabular overview of some previous work in multiprocessor robot controllers utilizing coarse grained task decomposition. Table 2.1 compares a few multiprocessor robot controllers developed using the heterogeneous system model, and Table 2.2 outlines controllers constructed with homogeneous architectures. The tabular comparison is balanced by a short discussion of the particular merits of each system including task partitioning methods, hardware configurations, and Real-Time Operating System (RTOS) support (if any).

The tables compare hardware components such as the host system (if present), real-time processors and real-time bus type. The methods used for IPC (interprocess/interprocessor communication) – shared memory (SM), message passing (MP), or both, are indicated for each system. Real-time support, either via a RTOS, function libraries, or specialized system hardware are characterized. The language used for the implementation of real-time programs is also listed.

The entries in both tables proceed in chronological order of reference material used in the literature survey. Entries in the same year are organized alphabetically by author name. The presence of question marks in a table entry indicates that the characteristic could not be determined from the literature reviewed.

### 2.3.1 Heterogeneous Controllers

The heterogeneous controllers surveyed all exhibit the characteristic of lack-

| Ref. | Name | Host | RT Proc. | Bus | IPC | RT Supp. | Lang. |
|---|---|---|---|---|---|---|---|
| [BWMJ88] [MWB89] | RIPS | Sun 3/140 | TMS320C25 CORDIC | VMEbus Private bus | SM MP | Hardware Software libraries | GNU C GNU C++ Assembler |
| [TIT89] | | 80286 | $\mu$PD77230 8086 | PC Bus | SM MP | | ?? |
| [And89] [And90] | | Sun 3/260 | 680x0 JIFFE | VMEbus | SM | RT host | C |

Table 2.1: Heterogeneous multiprocessor robot controllers.

ing a definable real-time operating system. This can in part be attributed to the relative difficulty in creating an operating system that would support the various types of processors, and yet provide the speed necessary for real-time robot control. In most of the above cases, the need for a real-time executive is eliminated by the system design itself and the use of special hardware expressly designed for the task at hand. As the hardware design becomes more specific, the need for a system executive to administer system resources diminishes. However, with the use of hardware designed for specific applications, the flexibility of the system suffers. Applications not well suited for the hardware platform suffer a loss in efficiency in the final implementation [BWMJ88]. In systems where the hardware design is driven by task requirements, tasks must be distributed to the hardware designed for them.

The Robotics Instruction Processing System (RIPS) utilizes an architecture optimized for the control of a robotic manipulator, at the expense of efficiency in problems not well suited to the hardware structure [BWMJ88]. RIPS incorporates CORDIC processors (for trigonometric calculations) and 32-bit ALUs and multipliers in a Robotic Processor which is used to calculate the kinematics and dynamics of a manipulator. A TMS320C25 DSP is utilized as the servo controller to provide high input/output rates (a strength of DSPs). A second TMS320C25 is used in the I/O module to provide high

speed interprocessor communication facilities.

In the RIPS controller, the programming language as well as the selection of hardware is driven by the requirements of each layer in the system. At the hardware interface level, where speed is a concern, assembly language is used. This layer involves reading sensor data, evaluating control laws and outputting motor torques. At the trajectory generation, inverse kinematics, and inverse dynamics level the C programming language is used. Programming in the task planning level is accomplished with C++, an object oriented programming language.

The system outlined in [TIT89] uses the $\mu$PD77230 DSP as a vector/matrix calculation engine. An 80286 based microcomputer is used as a user interface platform and to provide system timing constants. The DSP handles position/force control, coordinate transformation, sensor compensation, and monitoring of manipulator status. An 8086 processor for each manipulator joint handles joint velocity or current feedback control and joint status monitoring.

The JIFFE processor is a specialized scalar supercomputer architecture developed specifically for robot control and is targeted at performing numerical calculations at an extremely high rate (20 MFLOPS). The JIFFE processor may be used as either a coprocessor to a real-time host CPU, or used as a general purpose processor itself. In [And90], three JIFFE processors act as coprocessors to a real-time host CPU. A control JIFFE performs trajectory generation, kinematics, force or compliance control, dynamics compensation, and simple (or advanced) servoing algorithms at a 1 KHz rate [And90]. A second JIFFE is used to presimulate a motion before its execution to check for collisions or incompatibilities with the actual manipulator dynamics. The third JIFFE performs pixel-independent 3-D vision operations. The real-time host processor performs user interface and high-level task planning functions.

Very simple communication protocols based on predefined data transfer direction and verification of timestamps are used to avoid the use of semaphores. Programming of the JIFFE processor is accomplished via the C programming language.

### 2.3.2 Homogeneous Controllers

The homogeneous controllers reviewed are implemented with either the Intel 80x86 or the Motorola MC680x0 families of microprocessors. This is almost certainly due to the general popularity of these microprocessors and the ready availability of development information and tools for them. The Motorola family of microprocessors appears to be the most prevalent in the construction of multiprocessor robot controllers. This might be attributed to the use of Motorola's use of memory mapping for device interfaces, as opposed to Intel's more complicated I/O port addressing scheme. The notable exceptions to the use of the Intel or Motorola microprocessors are NYMPH [CFAB86], SPARTA [IK88], the Yale XP/DCS system presented in [BWLK89, WK90], and TUNIS [SG89].

The NYMPH (Not Your average Multiprocessor Hack) system possesses NSC32010 DSPs intended for raw control computational purposes. The 68010 CPU is actually a Sun 120 computer running V-System software with the VGTS window system [CFAB86]. The system is programmed in the C programming language with runtime libraries used to provide real-time communication and synchronization support. The NSC32010's handle all the real-time computing requirements for high speed floating point calculations, and the Sun 120 handles the user interface for the system.

SPARTA makes use of multiple IBM Hermes digital signal processors on

an IBM PC bus to provide real-time computation and I/O at sample rates in excess of 10 KHz [IK88]. Program development is accomplished through the use of the PLH language on an IBM VM/CMS mainframe. An IBM PC provides user interface and runtime support for the real-time system. No real-time executive is implemented for this system, however, some common operating system services are implemented for the DSPs in signal processor code.

The system presented in [BWLK89] and implemented in [WK90] utilizes the INMOS T800 transputer as the computing engine in the Yale XP/DCS dual-board real-time distributed control module. Multiple XP/DCS nodes may be incorporated into a system using the flexible and reconfigurable serial interconnection links supported by the T800. A real-time operating system is not required for this system, as the transputer offers hardware based real-time scheduling services and the OCCAM parallel programming language. The use of the unique interprocessor communication links eliminates many of the problems apparent in the other bus-based multiprocessor systems such as reconfigurability concerns and bus bandwidth limitations [BWLK89].

TUNIS can support up to ten NSC32016 based single-board computers in a master/slave relationship. One of the SBCs executes the TUNIS operating system which presents the user with a Unix-like interface. The remaining SBCs are managed by the central nucleus and may be used to execute standard Unix programs or real-time software. The system is programmed in Concurrent Euclid, which provides the monitor [2] construct for mutual exclusion, and signal and wait operations for synchronization.

The first implementation of the TUNIS controller used four NSC32016

[2]Theory and use of monitors is described in [Joe90].

SBCs (one master, three slaves) to provide servoing information to a PUMA's joint control hardware. The second implementation eliminates all PUMA control hardware and uses five NSC32016 SBCs to directly control the PUMA joint motors. One processor runs TUNIS, and a second provides for user functions. The third processor reads the PUMA encoders and calculates the kinematics. The last two processors split the task of calculating the dynamics of the PUMA manipulator. One calculates the dynamics for joints 1 to 3 and servos the PUMA joints, and the other calculates dynamics for joints 4 to 6.

The systems implemented with the Intel line of microprocessors includes the Robot Force Motion Server (RFMS) [PZ86], the controller presented by Kossman and Malowany in [KM87], the Robot Controller Test Station [MB89], and MRTA [Al-90]. The RFMS uses an Intel iSBC 86/30 (8086) with an 8087 floating point coprocessor to supervise the global variables and timing of the controller. An Intel iSBC 186/51 (80186) interfaces with the VAX 11/785 host via an Ethernet connection. A math processor, powered by an iSBC 286/12 (80286) with an 80287 math coprocessor, performs dynamics calculations, Jacobian updates, and matrix calculations. Each joint has an iSBC 86/30 (8086) with an 8087 coprocessor computing joint trajectories and joint torques. This controller is driven by interrupts and handshaking protocols and does not use a real-time operating system. All programming is done in the C programming language.

The controller presented by Kossman and Malowany [KM87] uses two microprocessors, an Intel iSBC 286/10 (80286) and an Intel iSBC 86/30 (8086), each with their corresponding floating point coprocessors. The 86/30 is used solely to communicate between the Multibus controller and the robot's joint controller CPUs (one 8085 CPU per joint). The 286/10 is used solely for path planning and control functions. Both CPUs run Intel's iRMX-86 operating

system. The 286/10 uses the Robot Control C Library (RCCL) and its underlying Real Time Control system (RTC) to provide a modular, extensible, and portable robot control and programming environment [KM87]. The C programming language was used, except for some of the RCCL math libraries which were rewritten in 80286/287 assembler to improve execution speed.

The RCTS is implemented using three Intel microprocessors. An Intel System 320 (80386) processor is employed for low-level control and sensor processing. Two Intel 80286 microprocessors are utilized for higher-level control and sensor processing. RCTS is organized in a hierarchal manner using a subset (the first three control levels) of the NASREM model. The lower levels of the hierarchy take precedence over the higher levels, and are therefore used to drive the system operation. For example, the motor controller module outputs a new joint torque at each sample of data from a sensor – the controller is driven by the sampler. The higher levels of control execute when the lower levels request new input [MB89]. The RCTS also allows easy replacement of modules. Each module is identical in structure, command set, status codes, and internal logic states. The system is supported by the iRMXII real-time operating system, and all programming is performed in C.

MRTA uses both loosely and tightly coupled interprocessor communication methods. MRTA is a multiprocessor system comprised of multiple multiprocessor modules. Each module is composed of eight 8086 microprocessors (with attendant 8087 coprocessors) connected to a specialized 8 port bus. The bus provides access to a shared resource which consists of memory, interrupt routing, and I/O for process control. Multiple modules of this type can be interconnected through member CPUs (at most, each processor may be connected to two 8 port busses). In this way, the system may be reconfigured to suit various scenarios. Shared memory is used for communication inside a mod-

ule, and intermodule communication is via message passing protocols. Data integrity is ensured through special arbitration hardware in the multiport bus and version numbers on all data elements. Task scheduling is achieved through the use of schedule generation software to generate and assign groups of tasks to a system's CPUs.

The system outlined in [DGI86] consists of a 68000 based workstation as host to multiple 68000 CPUs. The system utilizes a very simple real-time operating system, called ROS, consisting of a multitasking kernel, an event manager (a basic scheduler) and assorted software libraries. ROS also supports the concept of a device database to provide a measure of hardware independence. Physical devices are defined in the device database, and the system software uses this information to determine how the device is to be operated.

Harmony provides a multitasking, multiprocessing environment for real-time control. Development with Harmony has been accomplished on the Apple Macintosh, Vax/VMS, Vax/Unix, Wicat, and Sun platforms. Harmony has been ported to 68000, 68010, and 68020 boards as well as microVAX II [GWM+]. Harmony supports the execution and communication between multiple, concurrent tasks. It may be used with various development tools including windowed terminal emulators and debugging facilities.

The System for Implementing and Evaluating Robotic Algorithms (SIERA) utilizes a Multibus based system called the Real Time Servo System (RTSS) along with a loosely coupled Armstrong processor network [KWW87]. Both subsystems use MC680x0 family microprocessors, hence its classification as a homogeneous system. The hybrid architecture is used to combine the benefits of tightly and loosely coupled systems. The Real Time Servo System is used for real-time tasks, and the Armstrong processor array is used for non-real-time tasks, such as coordinating multiple robots or vision system tasks. A

limited form of multitasking is available on the RTSS in the form of up to two servo loops executing on any given Multibus processor. The servo loops are scheduled in response to interrupts, with higher priority given to the servo loop with the higher frequency.

The CONDOR and Chimera II operating systems are very similar in nature. Both use the VMEbus as the backplane for the real-time processors, and both utilize a Sun 3 workstation (Chimera II may also use a Sun 4) as a non-real-time host for development and graphical user interface duties. CONDOR and Chimera II also support the use of a bus to bus adapter to separate the real-time backplane from the non-real-time host. Filesystem services in both operating systems are provided by the workstation via the bus to bus adapter and a server process on the host. Some form of terminal emulation is provided for each system to facilitate communication between the real-time system and the host. Standard C libraries are included with each system, including one to perform matrix and vector calculations – a common requirement in robotic control systems. Interprocess communication via message passing and hardware independence through the use of Unix-like device drivers are supported in both systems.

CONDOR supports a maximum of eight Motorola 68020 processors on the real-time VMEbus. While not expressly a *multitasking* operating system, CONDOR does allow the scheduling of multiple servo loops using a fixed-priority scheduler. CONDOR provides debugging of processes on the slave CPUs via the emulation of the Unix *ptrace()* system call. Most Unix debuggers can be adapted to this system.

Chimera II supports a maximum of 128 single board computers within the real-time system. Chimera II offers a true multitasking scheduler which operates on a static/dynamic priority scheme called Maximum Urgency First

[SK91]. This scheduler, more complex than the servo loop scheduler in CONDOR, allows the use of dynamic tasks and modification of task priorities with no breakdown in the determinism of the real-time system. A task may be spawned on a CPU by any other task on that same CPU. Unfortunately, Chimera II lacks all but very rudimentary debugging facilities. The developer may either use the single-board computer's monitor, or place *printf()* statements within the C source code.

Hayati and Venkataraman describe a multiprocessor robot controller in [HV89] which can accept commands from either a six-axis teleoperator device or an autonomous planner, or both. This system utilizes 68020/68881 processors on VMEbus backplanes. A Sun 4/200 runs a modified version of UNIX OS 3.2 with real-time extensions and is used to calculate the kinematics and dynamics. The 68020 processors run the VxWorks operating system and perform servo-level operations at 1 KHz.

The Kumaran architecture, presented in [AGA90] uses six real-time Motorola MVME143 25 MHz 68030/68882 single-board computers on a VMEbus to run single-thread real-time processes. Unix Sys V.3 runs on a MVME147 20 MHz 68030/68882 SBC which provides RS-232, Ethernet, and SCSI interfaces. A second MVME147 is used to provide fast filesystem service to the real-time system. In this system, the Unix processor is used to develop the real-time software and monitor the execution of the real-time system via special Unix drivers. Multiprocessing and communication primitives are provided by library routines and Unix drivers. The fast filesystem is used to log data during the operation of the real-time system. After data collection is complete, the data is moved to the Unix filesystem for storage and analysis.

### 2.3.3 Concluding Remarks

Through examining the reviewed literature, it is apparent homogeneous systems tend to be employed in situations requiring more flexibility than their heterogeneous counterparts. The mapping of tasks onto processors in a homogeneous system tends to be characterized by a mapping of computing requirements over a range of available computing power instead of onto specifically designed hardware platforms. For example, contrast the rigid task allocation criteria for the Robotic Processor in the RIPS system with the criteria for allocating tasks under Chimera II. The Robotic Processor is intended to fulfill a narrow range of requirements, namely the calculation of manipulator kinematics and dynamics. The Robotic Processor would probably not make a very efficient platform for execution of a PID control algorithm. Similarly, the MC680x0 family of microprocessors supported by Chimera II are quite flexible in their applications, but suffer under the requirements of vector calculation. The gain in flexibility provided by a homogeneous architecture is offset by a cost in efficiency. General purpose microprocessors are not optimized for performance of typical robotic tasks, such as matrix and vector calculation tasks and high sample rate digital signal processing. The added computational cost of operating system overhead further reduces the effectiveness of microprocessors in a real-time multiprocessor configuration.

There appears to be two orthogonal directions of research in multiprocessor robot controllers using coarse-grained task division. The first direction, reviewed here under heterogeneous multiprocessor systems, is engaged in developing hardware dedicated to and optimized for the tasks inherent in a particular robot control scenario. These systems are characterized by the use of custom processors, such as CORDIC or JIFFE, and even the development of

VLSI chips for the real-time solution of inverse kinematic equations [LS87]. Systems of this type tend to be optimized for a particular control method or family of methods and are not very amenable to radical reconfiguration.

The other direction of research appears to be in the direction of flexible real-time systems utilizing the parallel processing power of homogeneous multiprocessor architectures. These systems stress ease of use, reconfigurability, and hardware independence. Most often, a RTOS is used to provide the first level of hardware independence, with function libraries providing increasing levels of abstraction. This field of research concentrates on optimizing the flexibility of the real-time environment for a wide range of robotic control tasks.

| Ref. | Name | Host | Processors | Bus | IPC | RT Supp. | Languages |
|---|---|---|---|---|---|---|---|
| [CFAB86] | NYMPH | VAX 11/780 | 68010<br>NSC32016<br>w/32081 | Multibus | SM<br>MP | Software<br>libraries | C |
| [DGI86] | | SM90 | 68000 | VMEbus<br>G64 | SM | ROS | C<br>FORTRAN<br>PASCAL |
| [PZ86] | RFMS | VAX 11/785 | 8086/87<br>80286/287 | Multibus | SM<br>MP | | C |
| [GWM+] | Harmony | Sun<br>Macintosh<br>Vax<br>Wicat | 680x0<br>MicroVaxII | VMEbus<br>Nubus<br>Multibus I<br>Multibus II | MP | Harmony | C |
| [KWW87] | SIERA | Sun 3/260 | 68000<br>Armstrong<br>processor | Multibus | SM<br>MP | RTSS | C<br>Assembler |
| [KM87] | | VAX | 8086/87<br>80286/287 | Multibus | SM | iRMX-86 | C<br>Assembler |
| [IK88] | SPARTA | IBM XT<br>IBM AT | Hermes | PC bus | SM | Lib. for<br>OS fcns. | PLH<br>C |
| [NSH88]<br>[NSH89] | CONDOR | Sun 3 | 680x0<br>w/68881 | VMEbus | SM<br>MP | CONDOR | C |
| [BWLK89]<br>[WK90] | | IBM AT | INMOS<br>T800 | Serial<br>links | MP | Hardware | OCCAM |
| [HV89] | | Sun 4/200 | 68020<br>w/68881 | VMEbus | SM<br>MP | VxWorks | C |
| [MB89] | RCTS | | 80286<br>80386 | Bitbus | SM<br>MP | iRMXII | C |
| [SG89] | TUNIS | | NSC32016<br>w/32081 | Multibus | SM | TUNIS | Concurrent<br>Euclid |
| [SKHK89]<br>[SSK89]<br>[SSK92]<br>[SSK90] | Chimera II | Sun 3<br>Sun 4 | 680x0 | VMEbus | SM<br>MP | Chimera II | C |
| [AGA90] | Kumaran | 68030<br>w/68882 | 68030<br>w/68882 | VMEbus | SM | Software<br>libraries | C<br>Assembler |
| [Al-90] | MRTA | IBM PC | 8086-2<br>w/8087 | Multiport<br>bus | SM<br>MP | Pre-<br>scheduling | ?? |

Table 2.2: Homogeneous multiprocessor robot controllers

# Chapter 3

# Design Methodology

## 3.1 System Requirements

In designing the robot controller described in this thesis, it is necessary to decompose the required functionality of the system into a set of concurrent tasks complete with the required communication pathways and protocols. It is then necessary to map this task structure onto the capabilities and features of the chosen real-time operating system and the supplied hardware platform.

The requirement specifications for the MRC are:

1. The controller must provide a convenient and flexible environment for the testing and evaluation of control algorithms for the PUMA 560 Industrial Robot. It must provide a reasonable level of robustness in terms of error identification and handling. It should be amenable to changes in system hardware configuration in that minimal effort should be needed in modifying code to deal with a change in hardware configuration.

2. It should be possible to select different control algorithms for testing without having to waste time recompiling the controller from scratch.

The system modules should therefore be as independent as possible to minimize the modules that need to be recompiled upon a major configuration change.

3. Function libraries for the interaction of control algorithms with hardware should insulate the user from the details of using the hardware and allow the use of floating point values (i.e. radian and voltage measurements) as an integral part of the control algorithm.

4. Use of a multi-windowed display should be made in order that software execution on each of the processors may be monitored in a convenient manner. Utilities such as real-time data logging and a graphical display of logged data should be present to aid in the evaluation of control algorithms.

## 3.2 Design Method

The design method chosen for this project is known as DARTS (Design Approach for Real-Time Systems) [Gom84]. This method yields a modular system with minimized intertask coupling and well-defined interfaces. This eases the interprocess/interprocessor communication overhead, and aids in simplifying future modifications to the system. The MRC's task hierarchy was loosely modeled after that of the NASREM model.

The design process begins with the functional requirements of the system given in the previous section. A traditional DFD (Data Flow Diagram) representation of the system is then analyzed to extract the major information flows and transformations that occur in the real-time system. The DFD is then structured to indicate the grouping of transforms into tasks based on the

following design criteria [Gom84];

- **Dependency on I/O.** A transform needs to be a separate task if its execution speed is dictated by that of the I/O device with which it interfaces.
- **Time-Critical Functions.** A transform that is time-critical needs to run at a high priority and hence as a separate task.
- **Computational Requirements.** A computationally intensive operation can run as a lower priority task.
- **Functional Cohesion.** Transforms that perform a closely related set of operations should be grouped together into a single task to cut down on intertask communication overhead.
- **Temporal Cohesion.** Transforms that perform their functions at the same time should be grouped into the same module.
- **Periodic Execution** Transforms that need to be executed periodically should be grouped into separate tasks.

The various tasks must then be mapped onto the available computing hardware. Constraints imposed by the real-time operating system chosen to support the multiprocessing requirements of the system must be taken into consideration as well.

## 3.3 Data Flow Analysis

The Data Flow Diagram (DFD) for this system is developed as a series of diagrams, each succeeding level of which describes the system in greater detail

than the preceding one. Each bubble in the DFD represents a transformation performed on information by the system software. The lines represent the flow of information with the arrows on the lines denoting the direction of flow. Data stores are represented by two parallel lines, and external entities (such as users) are modeled by blocks.

The first-level DFD is shown in Figure 4. This DFD represents the flow of data between all elements of the system, and is the starting point of the entire design, being derived directly from the system requirements statement. Commands are accepted from the user, and feedback is provided to the user in the form of a screen display. The manipulator is actuated via the application of voltages to the Unimation power amplifier assembly, and the outputs of the joint encoders, the joint pots, and the amplifier status lines are fed back into the controller. Data is logged to a user-specified file on the host workstation and may be retrieved for display and analysis. Text configuration files are used to provide easily modified configuration information for key aspects of the controller.

This initial Data Flow diagram is then reduced into component data flows and transforms which represent the planned flow of information through the robot controller. Once the DFD has been expanded to represent the system at an acceptable level of complexity (usually at the point that transforms begin to take on discrete and easily realized behaviors) the next stage of the DARTS design may be applied.

## 3.4 Task Structuring

The task diagram for the system is developed by applying the transform grouping criteria described in Section 3.2 to the final system DFD. This final DFD is

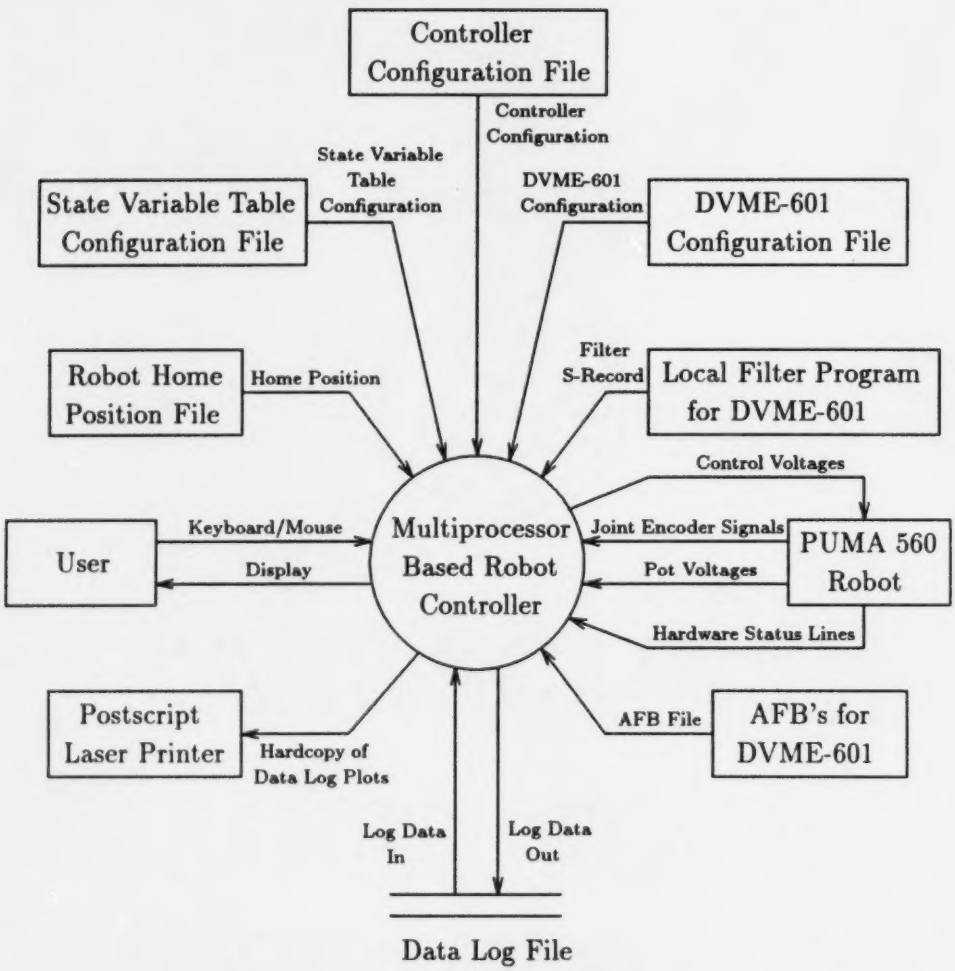


Figure 4: The top-level DFD for the MRC.

not presented here, as it is derived from the first-level DFD, and the relevant transforms and data flows are represented by the Task Diagram.

All transforms dealing with acquiring user input, performing error checking on this input, and displaying system responses to user input are grouped under the User Interface task in accordance with the Functional Cohesion criterion.

Transforms that receive the validated and preprocessed user commands and process these inputs into actions of the real-time system are grouped into the Robot Manager task using the Temporal Cohesion criteria. In this way, a user's command will be processed as soon as possible after it has been sent to the Robot Manager. Since the Robot Manager is responsible for starting and shutting down the system, it is also made responsible for periodically examining the state of the software and hardware modules to ensure that nothing unexpected has happened. The system is shut down gracefully if problems with either the hardware or software have occurred.

The control algorithm is structured as a separate time-critical task due to the fact that it must execute at regular intervals without fail. A strong case for these transforms being grouped as a task come from the Time-Critical and Periodic Execution transform grouping criteria.

The Motion Generation transforms must execute at a regular interval to supply the control algorithm with a block of joint setpoints at the setpoint update rate. These setpoints describe the motion of the manipulator through space, and are calculated to give a smooth motion from start position to finish, with all joints completing their motion at the same time. For this reason, these transforms are grouped together into their own task.

The data logging functions of the system must occur at periodic intervals that may or may not coincide with other periodic rates elsewhere in the system. If the system becomes computationally overloaded, it may be necessary to

reduce the rate of data logging to balance the load on the system's processors. Hence the data logging transforms are grouped into a separate task.

Since the sampling of the robot joint potentiometers and hardware status lines is performed on a separate CPU card (on the DVME-601 board), these transforms are grouped together into a separate task.

The symbols used in constructing the task diagram are shown in Figure 5. A task is denoted by a bubble containing the name of the task. External entities, such as the user, the robot, the Postscript laser printer, and the configuration files, are shown as rectangular blocks. Data stores, indicated by two parallel lines, are used in this project to represent state variable tables and data log files.

A tightly coupled queue symbol is used in cases where a message from one task requires a response or acknowledgement from the receiving task. This is important in cases where it is desirable to see the result of one message before another is allowed to be sent. A loosely coupled queue is a standard first-in-first-out buffer, often used in cases where messages are queued up pending processing by the receiving task. Single element queues are used to model cases where a variable contains a state or result, periodically updated by one task, available for use by another task. Software signals are denoted by the 'lightning bolt' symbol. Simple data paths, such as the input of a configuration file, are shown by lines with arrows indicating direction of data flow.

The complete task diagram is shown in Figure 6. This diagram reflects the software task structure and interprocess communication topology used in the final implementation of the MRC. This task diagram only indicates the relationships between the periodic portions of the system software. The initialization operations, necessarily sequential in nature, are not shown.

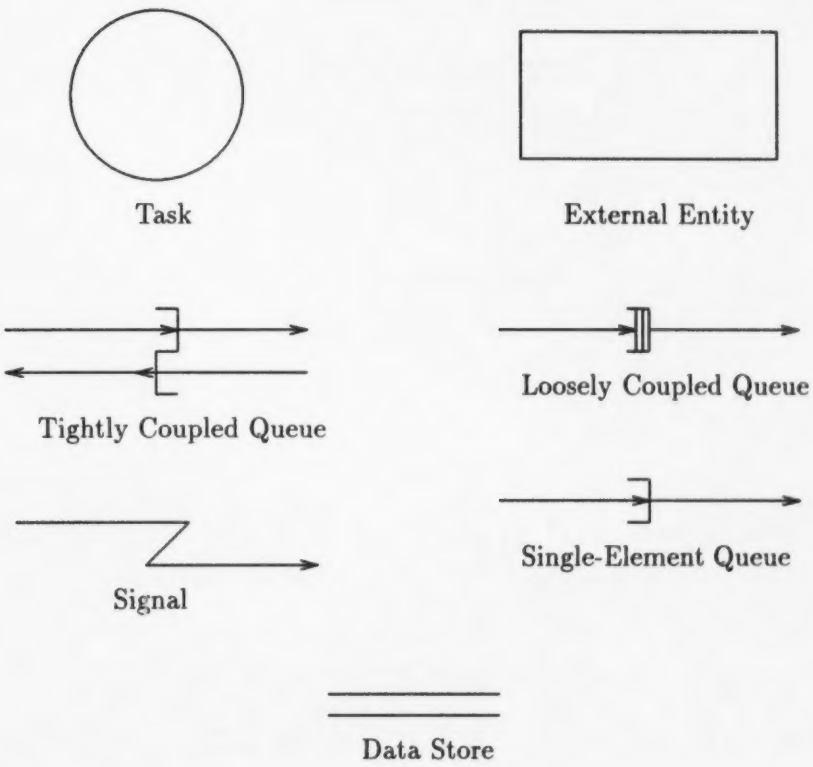


Figure 5: Communication and Synchronization Symbols.

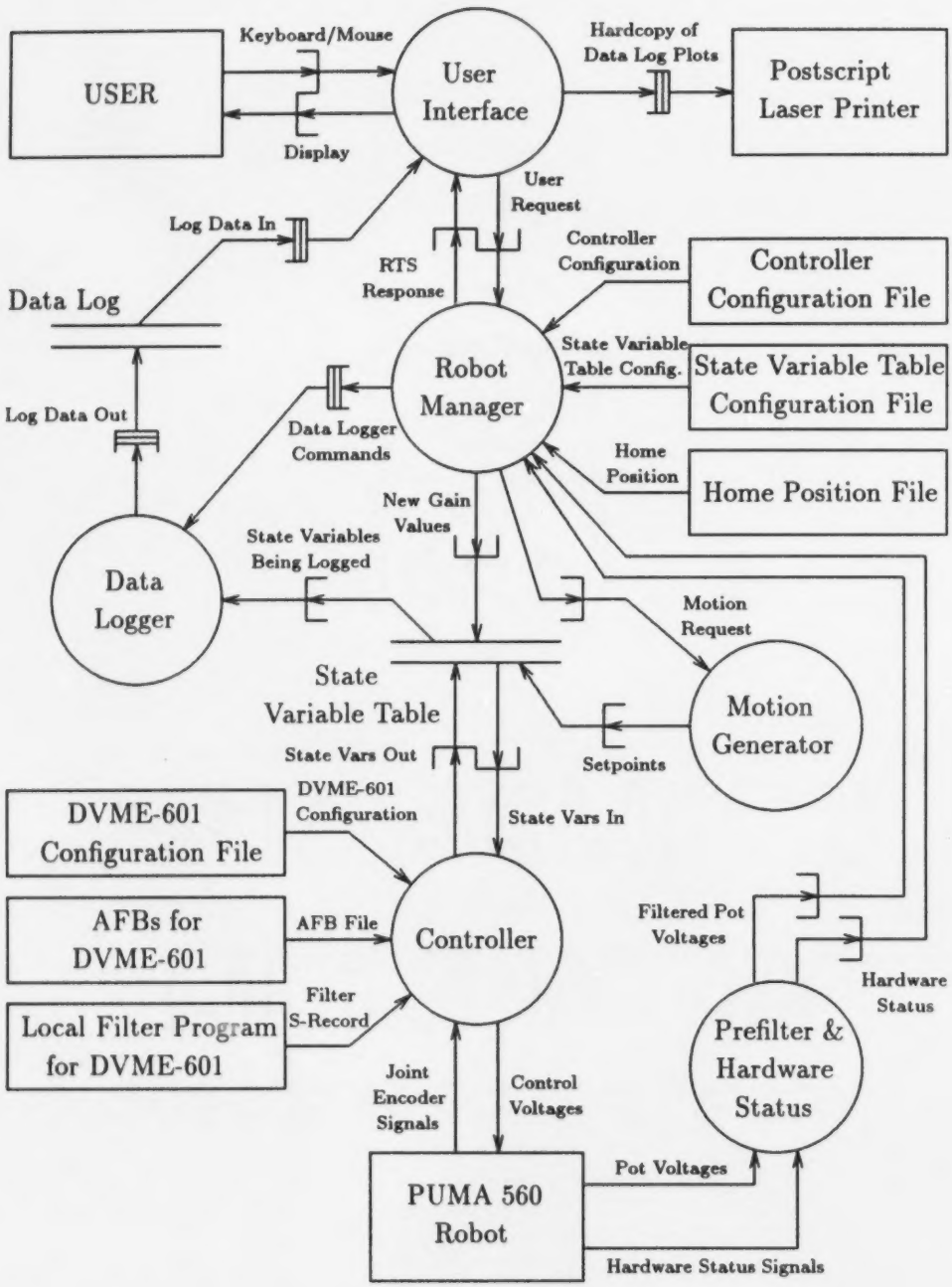


Figure 6: Task Diagram for the MRC.

## 3.5 Mapping of Modules to Hardware

The next step is the mapping of the software modules onto the available hardware. Valuable criteria for this stage are the suitability of a particular platform for a given task, and the minimization of VMEbus bandwidth usage. Bandwidth on the VMEbus may be conserved by assigning tasks which communicate at a high rate to the same RTPU, thereby keeping the communication local and not loading the VMEbus.

The reason for choosing the DVME-601 Analog to Digital converter for the tasks of sampling the robot joint potentiometers and providing a hardware status code is straightforward. The potentiometer and status line voltages are sampled directly by this board, and the capability exists for these signals to be processed by the DVME-601's onboard MC68010 microprocessor.

The SUN 3/160 is the logical choice for the User Interface Module, since this module will spend most of its time waiting on the user for input. The standard features of the SunOS operating system are more than adequate for this task, and the processor cycles on the RTPUs can therefore be conserved for more time-critical tasks.

The Controller task is relegated to its own Ironics IV-3220 SBC; the 'servo' RTPU. This decision is made because this task needs all the CPU power it can acquire. The absence of tasks other than the obligatory Chimera II servo task management process on this CPU allows more complex control algorithms to be implemented than if processor time is split among more tasks. The state variable table is implemented on the same RTPU as the Controller task. This helps conserve VMEbus bandwidth by not loading the VMEbus with the Controller task's state variable table accesses which occur at a rate higher than any other task in the system.

This leaves only the remaining Ironics IV-3220 (the 'master' RTPU) to provide a computing platform for the Motion Generator task, the Data Logging task, and the Robot Manager task.

## 3.6 The Real Time Operating System

It is necessary to select a Real-Time Operating System (RTOS) with the ability to administer the concurrent multiple processes required to implement the design. It is also desirable for the RTOS to supply such features as interprocess communication and synchronization, mutual exclusion, a filesystem, support for custom equipment (such as the VMEbus Encoder board), an easy to use development environment, and some form of support for multiprocessing configurations.

The operating system selected to support the MRC is the Chimera II Real-Time Programming Environment created at Carnegie Mellon University. Chimera II is classed as a *local* operating system. In other words, it uses another operating system to provide certain services or features. Chimera II uses the UNIX operating system running on a host workstation to provide a filesystem and a development platform for the real-time system.

Chimera II provides a number of useful features and tools for the development of sensor-based controls applications [SSK89, SSK92];

- A UNIX workstation provides an advanced programming environment and filesystem along with the convenience of readily available window environments.

- Multiple, commercially available general purpose CPUs are supported along with the requisite interprocessor communication and synchroniza-

tion primitives.

- Special purpose processors and I/O devices are easily integrated.
- A multitasking real-time kernel with user definable and selectable real-time schedulers along with a deadline failure mechanism provides real-time multitasking.
- A servo task management facility may be used to provide a convenient platform for applications utilizing multiple periodic processes.
- Reconfigurability is obtained through the use of reconfigurable state variable table facilities and a library for the use of text configuration files.
- Standardized interrupt and exception handlers are written in the C programming language.
- Access to I/O devices, special purpose processors, and the host filesystem is transparent through standard UNIX system calls.
- Hardware independence is achieved through the use of a *virtual machine layer* which insulates the application programmer as much as possible from the details of the hardware.

The features of Chimera II v1.11 are discussed in greater detail in Volume 2 of this project. More detailed information on Chimera II may be found in [SKHK89, SSK89, SSK90, SK91, SSK92, SSK91, Stea, Steb].

Although Chimera II provides many features useful for a project of this type, there were several areas in the project where difficulties arose due to local hardware compatibility problems and deficiencies in Chimera II.

Chimera II supports the use of VMEbus bus extenders such as the Bit3 Models 411, 412, and 413. These adapters are used to isolate the real-time bus

from the non-real-time host workstation. It was found that the Model 411 was not satisfactory for our purposes, since this card would only allow mapping of A24 (24-bit) address spaces between the real-time and non-real-time busses. Due to restrictions placed on available memory windows in the Sun's A24 space [Sun89] and available A24 base addresses for the Ironics RTPUs [Iro88], no more than one Ironics RTPU could be accommodated in the A24 space left available by Sun. There was no problem using the Model 412 as this uses A32 (32-bit) addressing which provides more than adequate memory space for the needs of Chimera II.

Chimera II uses the concept of *remote devices* to enhance its multiprocessing features. Under Chimera II each device is *owned* by one of the system's RTPUs. The remote device feature is meant to allow one RTPU to use a device owned by another RTPU. Unfortunately, there are a number of bugs in Chimera II v1.11 which render this feature unusable. The first problem involves the device lookup table which is initialized with an incorrect list of remote devices. This prevents proper remote device access. The bug was detected, fixed on the local system, and the patch sent to Carnegie Mellon University.

The second problem, also detected on the local system, involves the address translation required to convert the pointer information from one RTPU to an equivalent, accessible form for the second RTPU. This operation does not work correctly in Chimera II v1.11, and the remote device feature is consequently unusable. A workaround has been devised for the MRC wherein each RTPU must own any Chimera II device that it wishes to access. This is not an optimal solution, as it makes it difficult to implement fast mutual exclusion in a device driver. Both problems will be fixed in future releases of Chimera.

The state variable table facilities of Chimera II allow the creation of a global state variable table, sharable between multiple processes. One advantage of

the global state variable table approach is that it becomes much easier to create a controller consisting of replaceable modules. All support and user-interface software can be written independently of the type of control algorithm or setpoint generator actually used. The replaceable modules may then be written with a specific design in mind, and may simply be linked with the pre-compiled support and user interface routines at compile-time. At run-time, a state variable table configuration file will tell the system how to set up and use the state variable table.

The ease of implementing a reconfigurable system with the state variable table mechanism comes at a cost of increased complexity of the supporting software. It is necessary to write the support and user interface software in a general enough manner that it is able to handle all data types supported by the state variable tables themselves. This is especially true in cases such as data loggers or user interface tasks where the task must first attach to an arbitrary state variable table, and then proceed to process the required variables in the proper format.

State variable tables resident on the real-time system are not available for use by processes on the host workstation. If the host were allowed to access the state variable table on the real-time system, the non-real-time nature of the UNIX workstation would render the real-time system nondeterministic. For example, consider the case of a process on the UNIX workstation accessing the state variable table and obtain mutual exclusion lock on the data. If the workstation process was then swapped out before it could release the lock on the state variable table, the real-time tasks would be left waiting for the workstation process to be swapped back in to release the state variable table. This restriction of the use of the state variable table makes it difficult to implement a user interface running on the host that allows the user to interactively

modify state variable table values.

The MRC employs the Real-Time System Interface (RTSI) library to combat the aforementioned pitfalls in the use of Chimera II's state variable table mechanism. The RTSI library encapsulates much of the complex self-configuration code and provides extensions to the state variable table mechanism allowing access to state variable table data from the host workstation. This extension is covered in detail in Section 5.4.

The ability to utilize a higher level of debugging for the development phase of this project would have proved extremely useful. Chimera II has minimal debugging support in the form of *kprintf()* or *printf()* statements, as well as some memory allocation debugging routines that print out information on the system memory allocation tables. The user may also use the Ironics IMON Monitor to debug compiled real-time programs running on the Ironics RTPUs. The use of source-level debugging packages such as *gdb* is not supported in Chimera II.

## 3.7 Local Configuration of Chimera II

The first step in implementing the design of the robot control system is to successfully install and configure Chimera II for use on the Sun 3/160 workstation. The configuration file */etc/chimera.config*, which resides on the host workstation, specifies the various hardware components of the system, their addresses in the VMEbus addressing spaces (A16, A24, or A32), and the ownership of device drivers that are used by the real-time system. The configuration file for this project is given in full in Appendix A.3. Please refer to Chapter 2: Installation Manual in the Chimera II Program Documentation for a more detailed description of this file.

The configuration file for this system identifies three processors, the Sun 3/160 host as "robo2", the first Ironics IV-3220 SBC as the "master" RTPU, and the second Ironics IV-3220 SBC as the "servo" RTPU. The configuration file also indicates that the host is connected to the real-time VMEbus chassis holding the RTPUs via a Bit-3 Model 412 VMEbus-to-VMEbus adapter. The file specifies the addresses of the A/D converter, the D/A converter, and the VMEbus Encoder Board. The RTPU ownership of the device drivers for each of these interface cards is also indicated.

## 3.8 Assignment of Task Priorities

The Motion Generator, Data Logger, and Controller tasks all utilize Chimera II's servo task manager, or servo server, to provide real-time execution management. Please refer to Chapter 6: Reconfigurable Systems in the Chimera II Program Documentation for a complete description of this facility. In order to ensure that critical tasks on a RTPU receive the computing resources they require ahead of less critical tasks on the same RTPU, it is necessary to assign priorities to tasks executing on the real-time system.

Neither the User Interface task nor the Prefilter and Hardware Status task require task priorities to be set. The User Interface task executes on the Sun workstation and therefore does not require real-time scheduling. The Prefilter and Hardware Status task is the sole task running on the DVME-601 and does not use the servo server facility, therefore it does not require priorities.

The Robot Manager task is a low priority task which processes messages from the User Interface in spare CPU cycles on the 'master' RTPU. This task is involved in accepting user input from the non-real-time system, and routing user requests to the various tasks in the real-time system. These actions can

be assigned to a lower priority level, if a slight decrease in response to user input is acceptable. In this case, since no real-time constraints are applied to user input, it is judged that this tradeoff is acceptable.

The Controller task, the Motion Generator task, and the Data Logger task all use the servo server mechanism. These tasks must be assigned scheduler criticality and user priorities as per the guidelines outlined in [SK91]. Since the servo server facility uses a subset of the Maximum Urgency First (MUF) scheduling algorithm [1] to implement a Highest Priority First (HPF) scheduling algorithm, task deadlines need not be specified, and the criticality and user priorities assigned will compose the scheduling priorities.

The critical set of tasks for this system is composed of the Controller task and the Motion Generator task. These constitute the set of tasks that must not fail in a transient overload situation. Assignment of user priority and criticality to the Controller task is straightforward. This task and the servo server task are the only processes executing on the 'servo' RTPU. The Controller task is a member of the critical set, and therefore is accorded higher criticality and user priority than the servo server default value.

The assignments for the tasks on the 'master' RTPU are a more complex case. The Robot Manager task is left to execute at the same criticality and user priority as the servo server task which executes there. The Motion Generator task is a member of the critical set and is therefore set to the highest criticality and user priority on the 'master' RTPU. The Data Logging task, while not a member of the critical set, is still a more important task than the Robot Manager, and is therefore set to a criticality and user priority between those of the Robot Manager and the Motion Generator.

---

[1] This will be upgraded to full MUF in a later versions of Chimera [SSK91].

The result of these task criticality and user priority allocations is that on the 'servo' RTPU, the Control Algorithm task will be the last to fail in order to preserve the generation of valid joint torques for the PUMA manipulator. The situation on the 'master' RTPU has the Motion Generator task failing last in order to protect the generation of valid setpoints for the Control Algorithm task. The Robot Manager task will be the first task to miss a deadline, but this is relatively unimportant since that task only deals with user interface functions anyway. The Data Logging task is the next to fail after the Robot Manager in a transient overload of the 'master' RTPU.

The user must take care to provide reasonable execution rates for the processes involved. If the execution rates are set too high, response of the Robot Manager begins to slow down to an unacceptable level, and the Data Logger task begins to miss data samples. The information on missed cycles for the real-time processes using the servo server facility are easily available from the user interface (please refer to the Software Users Guide in Volume 2). This information may then be used to choose appropriate execution rates for the three servo server tasks.

# Chapter 4

# Controller Hardware

## 4.1 Introduction

This chapter describes the configuration and modification of existing hardware necessary during the implementation of the MRC. The hardware aspect of this project involves assembling a hardware platform compatible with the Chimera II Real-Time Operating System. The hardware must also support the design decisions made with respect to the requirements specification outlined in Chapter 3.

A detailed overview of the configuration of the MRC's hardware components can be found in Appendix A. Memory maps for both the Sun 3/160 and the real-time VMEbus chassis are given, showing the assignments of memory space in both chassis. The Chimera configuration file used to describe the hardware setup to the operating system is presented for reference as well.

## 4.2 Design Issues

The hardware design phase of the project consisted mainly of determining the configuration of the various hardware components for operation with the Chimera II RTOS.

The controller hardware is physically divided into two main components, the RTS (Real-Time System) and the NRTS (Non Real-Time System). The RTS is used to perform all the time critical tasks such as providing digital control algorithms for the joint motor amplifiers, performing real-time data logging, generating the robot's path, and detecting hardware or software failures. The NRTS is utilized for tasks that lack hard real-time constraints, such as accepting input from the user, checking that input for errors, providing a graphical display of logged data, and providing a development platform and filesystem to support the Chimera II operating system. The two systems are connected by the Bit3 VMEbus to VMEbus interface which helps to isolate the RTS from the NRTS. The block diagram of the system hardware is shown in Figure 7.

## 4.3 Non-Real-Time System

The Sun 3/160, as the NRTS platform (also referred to as the *host*), provides a large color graphics monitor and NFS access to 1.5 GB of online disk storage space. This workstation also contains its own local hard disk which is used for local virtual memory space and basic SunOS system programs. Note that an Ethernet bridge was installed to isolate the Robotics Lab LAN from the campus Ethernet backbone. This keeps the lab's Ethernet traffic, made heavy by the use of the X Window System, localized so that it will not load the

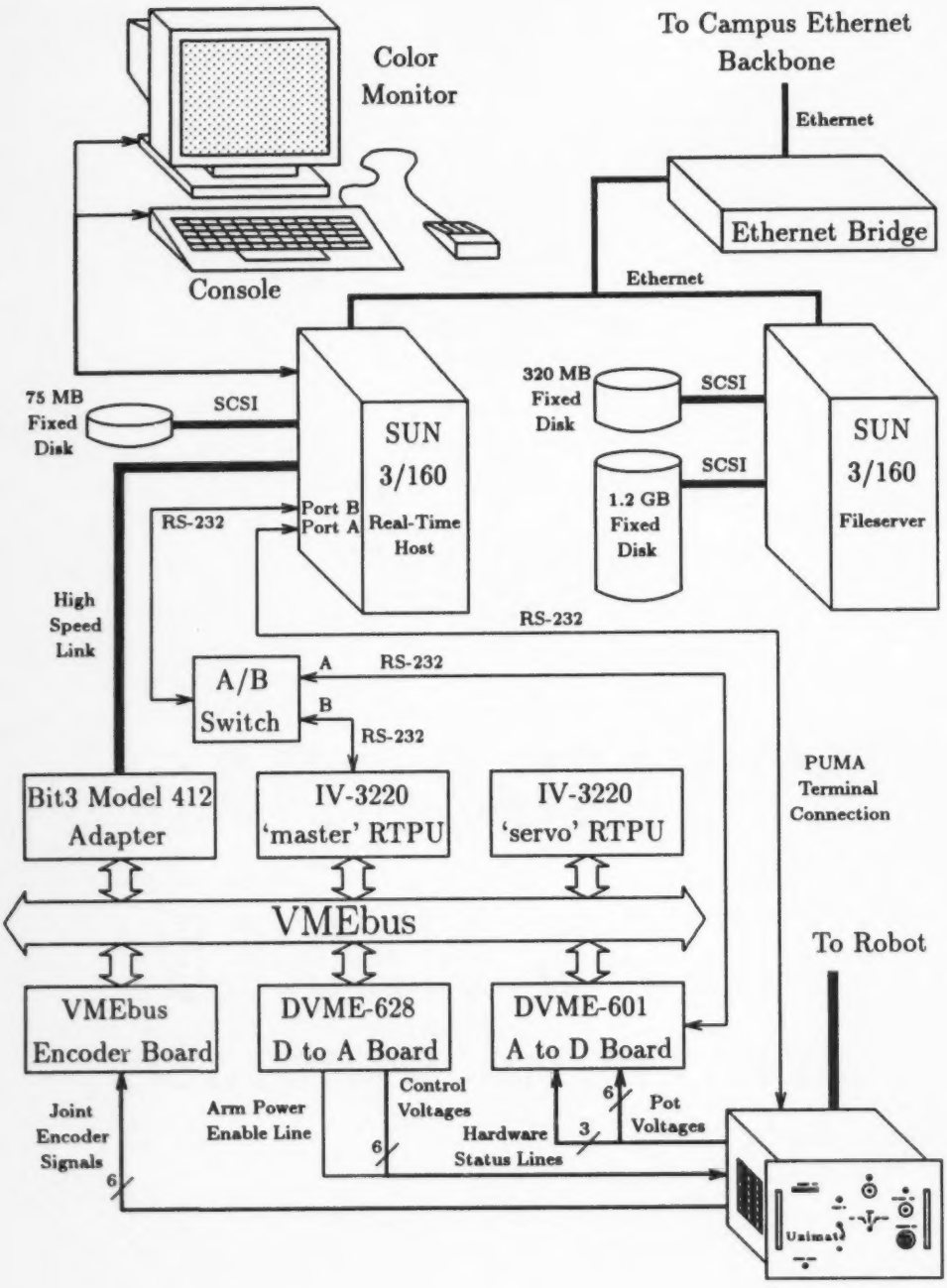


Figure 7: Hardware for the Multiprocessor Based Robot Controller

campus backbone.

The second Sun 3/160 shown in Figure 7 is used as a fileserver for the host workstation. Its large capacity hard drives store the executables and source files for the real-time system as well as the balance of the SunOS system programs and files needed for the operation of the UNIX workstation.

The host workstation is also used to provide virtual terminals for accessing the PUMA RS-232 terminal port, the DVME-601 RS-232 terminal port, and one of the IV-3220 RS-232 ports. The virtual terminals, created for the MRC under the X Window System, provide 'dumb terminal' emulation for equipment connected to the two workstation serial ports. The terminal program itself may be invoked to attach a virtual terminal to either port A or port B.

Port A is connected directly to the Unimation Control Computer terminal port. The virtual terminal program on the Sun workstation provides access to the VAL-II operating system when the VMEbus controller is switched out and the PUMA is being used with the stock control computer. Port B is connected via an A/B switch to either the DVME-601's serial port or a serial port on the 'master' RTPU. This connection provides access to either the DVME-601 or the IV-3220 onboard monitors through a convenient window on the Sun workstation.

The Sun 3/160 requires the installation of a Bit3 Model 412 VMEbus to VMEbus adapter card in order that it may be connected to the Real-Time VMEbus via a similar card installed there. The Bit3 cards are configured in master/slave mode. The master Bit3 card (installed in the Sun backplane) maps the address spaces defined on the slave (installed in the Real-Time VMEbus chassis) into the memory space of the host workstation's VMEbus. A small portion of this memory is then mapped into Sun's Direct Virtual Memory Access (DVMA) feature. The DVMA space is never swapped out by SunOS and

can therefore be used for communicating with the Chimera II kernels running on the Ironics boards [SSK89].

The Bit3 cards may also be equipped with dual-port RAM for use in communications between the two separate VME busses. The MRC has 32KBytes of dual-port ram on the slave Bit3 card available for use by Chimera II.

Please refer to Appendix A.1 for the system memory map and to Appendix A.2.1 for the local configuration of the Bit3 Model 412's onboard jumpers for both the master and slave boards.

## 4.4 Real-Time System

The RTS is implemented on the external VMEbus chassis. Its hardware components consist of the two Ironics IV-3220 Single Board Computers in conjunction with the Datel DVME-601 Analog to Digital Converter, the Datel DVME-628 Digital to Analog Converter, the VMEbus Encoder board, and the slave Bit3 VMEbus to VMEbus adapter. The RTS is responsible for all real-time tasks required of the controller such as setpoint generation, data logging, and calculation of control voltages for the PUMA joint amplifiers. Please refer to Appendix A.1 for the memory map of the RTS, and to Appendix A.2 for the local jumper settings for these cards.

The existing controller used the TUTOR board to drive the VMEbus clock (SYSCLK) at 12 MHz. The Bit3 Model 412 used as system controller for the Real-Time VMEbus in this project drives SYSCLK at 16 MHz. This increase in speed requires that the DTACK delay jumper on the VMEbus Encoder Board be advanced from Position 0 (no delay) to Position 1 (one clock cycle delay) to compensate for the higher SYSCLK speed. The delay allows the VEB's address decoding chips to settle before the VEB responds to addresses

on the VMEbus.

If the DTACK delay is improperly set, the VMEbus may incorrectly respond to addresses presented on the VMEbus address lines. This can result in erratic behaviour of the VEB, the most apparent of which is sporadic resetting of the joint encoder counters. This manifests itself as erratic movements of the PUMA manipulator.

As originally designed, the VEB was inadequate for use with a sophisticated operating system such as Chimera II. The board could be set to respond to either of two *address modifier codes* [VME82]. The address modifier codes are used to distinguish between *types* of memory access. The two codes originally available (and therefore the two types of memory accesses available) on the VEB were;

- 0x39 - Standard Non-Privileged Data Access
- 0x3D - Standard Supervisory Data Access

The Chimera II kernel uses the Standard Non-Privileged Data Access for user reads and writes of a device, but it uses the Standard Supervisory Data Access for initializing the device. Since the VEB could only be set to one or the other type of access, either reads or writes of the VEB worked correctly, or the kernel initialization of the board worked correctly, but not both.

The VEB has been modified so that it accepts accesses in both Standard Non-Privileged and Standard Supervisory Data Access modes. The VEB jumper settings as well as the updated schematics of the VMEbus Encoder Board's jumper layout and address decoding are given in Appendix A.2.5.

# Chapter 5

# Controller Software

## 5.1 Design Goals

The implementation of software for the MRC consists of realizing the task structure design developed in the System Design (see Chapter 3). The key criteria for the software implementation are:

- **Hardware Independence.** The software should use the concepts of standardized device drivers and function libraries to ensure that the code may be ported to other platforms with a minimum of effort. Hardware independence also insures that changes in hardware do not automatically mean that all levels of code must be rewritten.

- **Flexibility.** The software must support easy modification of the controller and motion generation algorithms. This requires that modules not immediately concerned with these two tasks be as independent of their implementations as possible. In general, the software should not depend on the actual control algorithms used. All algorithm-specific information should be obtainable from configuration files.

- **Encapsulation of Data.** In order to improve the maintainability of the system, data and related functions should be encapsulated within libraries whenever possible.

Hardware independence and data encapsulation are achieved through the use of libraries and functions to provide levels of abstraction between the application programmer and hardware elements such as the A/D and D/A converters. Functions and associated data structures used for a common purpose are grouped together into libraries and their administrative details hidden from outside view. For example, all functions and data structures dealing with the communication between the Real-Time System and the Non-Real-Time System are grouped together into the RTSI set of functions.

Flexibility is provided by ensuring that all tasks use well defined interfaces, and their implementations are as independent as possible from any particular control algorithm. The state variable table, servo task management, and configuration file mechanisms of Chimera II are used extensively to achieve this goal. The state variable table mechanism provides a convenient, reconfigurable data store of controller state variables. Information on each state variable is easily obtained by other tasks and is used in self-configuration routines to ensure that each state variable is used in the correct manner. The servo task management facility provides simplified control of the controller's periodic tasks. Periodic tasks may be written in a well-defined manner, using a framework outlined in Chapter 6: Reconfigurable Systems in the Chimera II Program Documentation [SSK91]. The configuration files allow the user to change the operation of the MRC without recompiling the controller source code.

## 5.2 Software Module Description

The Controller, Motion Generator, and Data Logger tasks are all implemented using Chimera II's *servo server* facility. This provides the easiest and most convenient method for controlling the execution of these periodic tasks. Intertask communication is simplified as well, with most real-time intertask communication taking place via the state variable table mechanism. The remaining tasks in the system, the User Interface, Robot Manager, and Prefilter and Hardware Status tasks, do not utilize the servo server mechanism.

The communication link between the Robot Manager and the User Interface is implemented with a special purpose set of functions implemented using interprocess/interprocessor communication facilities of Chimera II. These functions, the Real-Time System Interface (RTSI) library, are described in Section 5.4.

### 5.2.1 User Interface

The user interface for the system is a command-line oriented preprocessor that may be run either on a plain text terminal, or in a separate window in the user's choice of windowing environment. A text based approach is chosen, since Chimera II provides a command-line parser in the form of the 'cmdi' library. This library allows a user to easily add and delete commands for a command-line based interface. Since this system is intended for use in a research environment, this feature was deemed very valuable. A complete description of all currently available commands may be found in the Software User's Guide in Volume 2.

The user interface employs the RTSI library for all communications between itself and the Robot Manager task. A client-server relationship exists

between the user interface and the real-time system. The user interface sends messages to the real-time system which services them in the appropriate fashion. All messages sent from the user interface require a reply from the real-time system to indicate the status of the action. This provides a means of message queue error detection for both the user interface and the real-time system.

### 5.2.2 Robot Manager

The Robot Manager task is responsible for the supervision of all other tasks in the system. It takes care of initialization and shutdown of all parts of the MRC. All errors are filtered back through the Robot Manager for handling. It also monitors the robot hardware and software for failures. Since the robot is automatically shut down on critical errors by the Unimation hardware, monitoring the joint amplifier hardware for errors is performed for informational purposes only. The Robot Manager monitors the state of the other processes through the return values of servo server function calls. The tasks themselves possess the ability to shut down and flag the servo server upon detection of an error. The status of the internal memory allocation table is also monitored by the Robot Manager as a safeguard against memory corruption. Upon detection of an error condition the Robot Manager will proceed with an orderly shutdown of the real-time system.

The Robot Manager also looks after servicing requests from the user interface. The requests are processed in a first-come first-served basis, and the applicable information or signals are routed to the appropriate tasks.

When the controller is started, the Robot Manager's initialization sequence methodically initializes the data structures and other tasks that compose the controller. The Robot Manager first initializes storage for internal data struc-

tures and processes any optional environment variables that have been set. The User Interface task is then started on the host workstation via a RTSI library call, the Controller Configuration file is processed, and the servo server context is created for the controller.

The Robot Manager then waits for the user to select the pre-linked control algorithm with which to control the robot arm. This selection controls the default rates for the Motion Generator task, the Data Logger task, and the Controller task. The State Variable Table Configuration file for the selected controller is then used to initialize the controller's state variable table. A RTSI library call is then used to send relevant information about the state variable table to the User Interface for error checking purposes.

The Robot Manager then initializes access to the VMEbus Encoder Board initialization mechanism and the DVME-601's pot voltages for use in calibration of the robot arm.

The last step before the Robot Manager enters its message-processing loop is the initialization of the Controller, Motion Generator, and Data Logger tasks. Once in the loop, the Robot Manager responds to and acts upon messages from the user interface.

### 5.2.3 Motion Generator Task

The Motion Generator task provides setpoints to the Controller task in order to realize motion of the robot manipulator. Two different types of motion generation were implemented in this module. The first type of motion, *blended motion*, is the type usually associated with robot joint motion. Blended motion consists of a series of setpoints calculated to move the robot's individual joints smoothly from an initial set of joint angles ($\theta_0$) to a final position ($\theta_f$) in a

given length of time ($t_f$). The second type of motion implemented was a very simple *step* input to the Controller task. Step motion is simply the application of a step input of a specified magnitude to the Controller task for use in testing the response of control algorithms.

In order to specify a motion for this system, a *motion block* is passed to the Motion Generator from the Robot Manager. This motion block consists of a set of goal joint angles (the $\theta_f$'s for each of the six PUMA joints), and a speed factor (specified as a decimal value from 0 to 1) which is a maximum desired joint velocity specified as a fraction of the absolute maximum velocity. The motion block is sent with an attached motion type which specifies the form the motion will take (currently only joint-interpolated motion and step motion are implemented). This format is meant to be general enough to characterize a motion for any type of positional reference motion generation algorithm, thus isolating the motion generation module from the rest of the software, and making it wholly replaceable.

At the beginning of each motion generation cycle, the Motion Generator checks an internal flag to see if a move is in progress. If a motion is in progress, the Motion Generator will proceed to calculate the next set of setpoints for the joint controller. If not, it will obtain the next motion block and type from the motion queue.

If the motion block type is a step, the magnitude of the step (contained within the motion block) is added to the current position of the joint being stepped. This value is then written to the setpoint state variable, and the Motion Generator returns to checking the input queue.

If the motion block type is a blended move, the Motion Generator must initialize the calculations that will describe the joint motion. The blended motion is constrained to having a velocity of zero at the beginning and end of

every motion. Each motion in this system is complete unto itself — there is no blending of queued motion requests.

These design criteria are met by the use of a third order polynomial blending function [Cra89]:

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{5.1}$$

Differentiating equation 5.1 gives the velocity:

$$\dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2 \tag{5.2}$$

Applying the velocity constraints at the endpoints, yields the following system of four equations:

$$\theta(0) = \theta_0 = a_0 \tag{5.3}$$

$$\theta(t_f) = \theta_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \tag{5.4}$$

$$\dot{\theta}(0) = \dot{\theta}_0 = a_1 = 0 \tag{5.5}$$

$$\dot{\theta}(t_f) = \dot{\theta}_f = 2a_2 t_f + 3a_3 t_f^2 = 0 \tag{5.6}$$

$$\tag{5.7}$$

Using these four equations, one may solve for the four unknowns $a_0, a_1, a_2, a_3$:

$$a_0 = \theta_0 \tag{5.8}$$

$$a_1 = 0 \tag{5.9}$$

$$a_2 = \frac{3}{t_f^2}(\theta_f - \theta_0) \tag{5.10}$$

$$a_3 = -\frac{2}{t_f^3}(\theta_f - \theta_0) \tag{5.11}$$

In order to apply equation 5.1 to the generation of setpoints for this particular system, it was necessary to make a few modifications in order to take into consideration maximum allowable joint velocities and accelerations, the

fact that the setpoint update function is a *discrete* function, not a continuous one, and minimization of CPU load.

For this motion generation algorithm, it is desired that all joints would complete their movements at the same time, yielding a smooth, fluid motion of the manipulator. It is also desirable that no joint exceeds the speed set by the user-supplied speed factor. To accomplish this, it is necessary to calculate the time it would take each joint to complete its motion given that each joint must not exceed its specified velocity or maximum allowed acceleration. The specified maximum velocity for a joint is given by:

$$\dot{\theta}_S = S_F \dot{\theta}_{MAX} \tag{5.12}$$

Where $S_F$ is the speed factor contained in the motion block, and $\dot{\theta}_{MAX}$ is the absolute maximum allowed velocity of the joint which is taken as the Unimation defined joint velocity at VAL-II's SPEED 100 [Uni85].

By differentiating equation 5.2 with respect to time, and setting this to zero, the time at which the maximum velocity of the joint occurs can be found to be:

$$t_{v_{MAX}} = \frac{t_f}{2} \tag{5.13}$$

By substituting 5.13 for $t$ in 5.2 the maximum velocity of the motion (at time $t_{v_{MAX}}$) is obtained:

$$\dot{\theta}(t_{v_{MAX}}) = \frac{3}{2t_f}(\theta_f - \theta_0) \tag{5.14}$$

This maximum velocity must be less than or equal to the user specified maximum velocity ($\dot{\theta}_S \geq \dot{\theta}(t_{v_{MAX}})$) for the limit on maximum joint velocity to be upheld. This enables the time of movement $t_f$ to be calculated from the specified maximum velocity ($\dot{\theta}_S$) and change in angular position ($\theta_f - \theta_0$) by

substituting the above inequality into equation 5.14:

$$t_f \geq \frac{3}{2\dot{\theta}_S}(\theta_f - \theta_0) \tag{5.15}$$

It is also necessary to take into consideration the maximum allowed acceleration for a desired joint movement. This is especially important in cases where the speed factor is large and the change in joint angle is small, as the joint then must undergo large accelerations over a small number of sample instants, and problems with joint tracking often result. The solution to this problem is to calculate the maximum acceleration of the move and compare this to a pre-defined limit (defined in the *globalPUMA.h* header file listed in Volume 2).

The acceleration of the joint is given by differentiating equation 5.2 with respect to time:

$$\ddot{\theta}(t) = 2a_2 + 6a_3 t \tag{5.16}$$

Substituting in the values of $a_2$ and $a_3$:

$$\ddot{\theta}(t) = \left(1 - 2\frac{t}{t_f}\right)\frac{6(\theta_f - \theta_0)}{t_f^2} \tag{5.17}$$

This is the equation of a line which, in the domain of interest ($t = 0$ to $t = t_f$), has a maximum magnitude at $t = 0$ and $t = t_f$. Therefore the magnitude of the maximum acceleration (since it is the same at beginning and end) may be written:

$$\ddot{\theta}_{MAX} = \frac{6|(\theta_f - \theta_0)|}{t_f^2} \tag{5.18}$$

This maximum acceleration magnitude is calculated using the $t_f$ obtained from equation 5.15. If the value is larger than that joint's maximum allowed acceleration, the value of $t_f$ is recalculated using the equation:

$$t_f = \sqrt{\frac{6|(\theta_f - \theta_0)|}{\ddot{\theta}_{MAX_{Allowed}}}} \tag{5.19}$$

Note that this then takes the determination of joint velocity out of the hands of the user in order to obtain adequate tracking of the joint motion. The values of $\tilde{\theta}_{MAX_{Allowed}}$ for the manipulator joints are obtained by estimation using the Unimation defined maximum acceleration times and joint velocities at VAL-II's SPEED 100 [Uni85]. The values are then fine tuned by trial and error for the specific joint.

The movement duration is calculated in the above manner for each of the six joints. The largest $t_f$ is chosen to be the $t_f$ for all joints in the current move. This ensures that all joints will stop moving at the same time, and no joint will exceed its specified speed.

The fact that the Motion Generator updates the setpoint every $T_u$ seconds can be used to determine how many cycles of the Motion Generator it will take to complete the desired motion. The number of cycles required is calculated by applying the ceiling operation to

$$N_c = \frac{t_f}{T_u} \tag{5.20}$$

By substituting $N_c$ for $t_f$ in the motion equations, the Motion Generator produces the desired motion over an integer number of cycles.

To minimize the computational load, the setpoint sequence is calculated once each setpoint update period during a move in order to spread the load of computing the setpoint sequence over the total number of update periods. The first setpoint period bears the heaviest load, as the values of $t_f, N_c$, and the coefficients of the polynomial (listed here as $b_2$ and $b_3$ for the discrete case) must be calculated. The calculations for the coefficients are performed in the following manner in order to optimize the operations for speed:

$$\Delta\theta = \theta_f - \theta_0 \tag{5.21}$$

$$t_f = \frac{\Delta\theta}{S_F\dot{\theta}_{MAX}} \tag{5.22}$$

$$N_c = ceil\left(\frac{t_f}{T_u}\right) \tag{5.23}$$

$$b_2 = \frac{3}{N_c^2}\Delta\theta \tag{5.24}$$

$$b_3 = -\frac{2}{N_c^3}\Delta\theta \tag{5.25}$$

The new setpoint is calculated as:

$$\theta(k) = \theta_0 + b_2k^2 + b_3k^3 \tag{5.26}$$

where $k$ is an integer counter supplied internally by the Motion Generator.

The updated setpoint is then written to the local copy of the setpoint state variable and will be copied into the global state variable table at the end of the present Motion Generator cycle.

### 5.2.4 Controller Task

It is possible for more than one control algorithm module to be linked to controller software at compile time. The control algorithm to be used is chosen upon controller startup from a list indicating all control modules linked into the controller. This choice determines the configuration of the state variable table and the Motion Generator module selected. However, only *one* of the controller modules may be chosen as the Controller task at any one time.

The controller task is designed as a replaceable module. Because of the fact that different control algorithms may require different input information from the motion generator, the selection of a particular control module on startup will select that control module's complementary Motion Generator task as defined in the *controller.cfig* file.

The control module implemented to test the MRC was a simple PID controller with a position reference supplied by the Motion Generator task via the state variable table mechanism. This controller also employs the state variable table to provide dynamically alterable proportional, integral, and derivative gains for online tuning purposes. Section 6.3 gives a more complete description of the controller algorithm.

### 5.2.5 Data Logger Task

The Data Logger module can be dynamically configured from the user interface to record the values of any of the controller's state variables stored in the state variable table. Each sample of the chosen state variable(s) is recorded along with a timestamp indicating the time the sample was taken.

The sampled data is recorded into a large (2 MB) buffer on the 'master' RTPU. These direct writes to memory allows the speed of the logging to be maximized and non-essential VMEbus accesses to be minimized. The price of this performance is a limit on the time window for logging operations determined by the number of state variables to be logged and the rate the state variables are sampled. The time limit for each logging operation is calculated and printed when the Data Logger module is started.

After the logging of data is complete, the entire buffer is dumped to a user specified file on the Sun 3/160's filesystem in a format suitable for further processing by the *gnuplot* graphics package. A small header is also written onto the file describing its source and what state variables were sampled. Also generated at this time is the stub of a *gnuplot* macro file which can later be modified to display the data in a specific manner using the *gnuplot* graphics package.

The Data Logger task is an example of a *dynamic task* which only comes into being for a period of time, and is stopped when its usefulness is ended. This can have adverse effects on a real-time system, as the load on the CPU increases when this task is started. This can cause real-time tasks to begin to miss deadlines as the processing power required by the multiple tasks exceeds that which the CPU is capable of supplying.

A failure of the Data Logger task has the effect of leaving holes in the sampled data should the Data Logger task fail. This is not as disastrous as it may first appear. Using the timestamp written for each data sample, it is possible to adjust for the missing data. Usually the missing point(s) are simply ignored by the graphics package used, and a straight line is used to approximate their locations. If the Data Logger's period is so low that it begins to miss cycles, its period may be increased to reduce the computational load on the 'master' RTPU to ensure that no cycles are missed. However, it is usually not necessary to sample so fast that the RTPU becomes overloaded.

### 5.2.6 Prefilter and Hardware Status Task

In order to smooth the sampled voltages from the PUMA's joint potentiometers, it is necessary to implement some form of prefiltering on the DVME-601 Analog to Digital Converter's onboard MC68010 processor. The filtering implemented is similar in form to that in Kruszewski's Controller; however there are key differences between the two approaches.

The filter implemented for this project is a four-point moving average filter with a sampling rate of 1000 Hz. The filter implemented in [Kru90] was a four-point batch filter also with a sampling rate of 1000 Hz (the sampling rate is programmed into the DVME-601's 68681 timer). The new filter produces

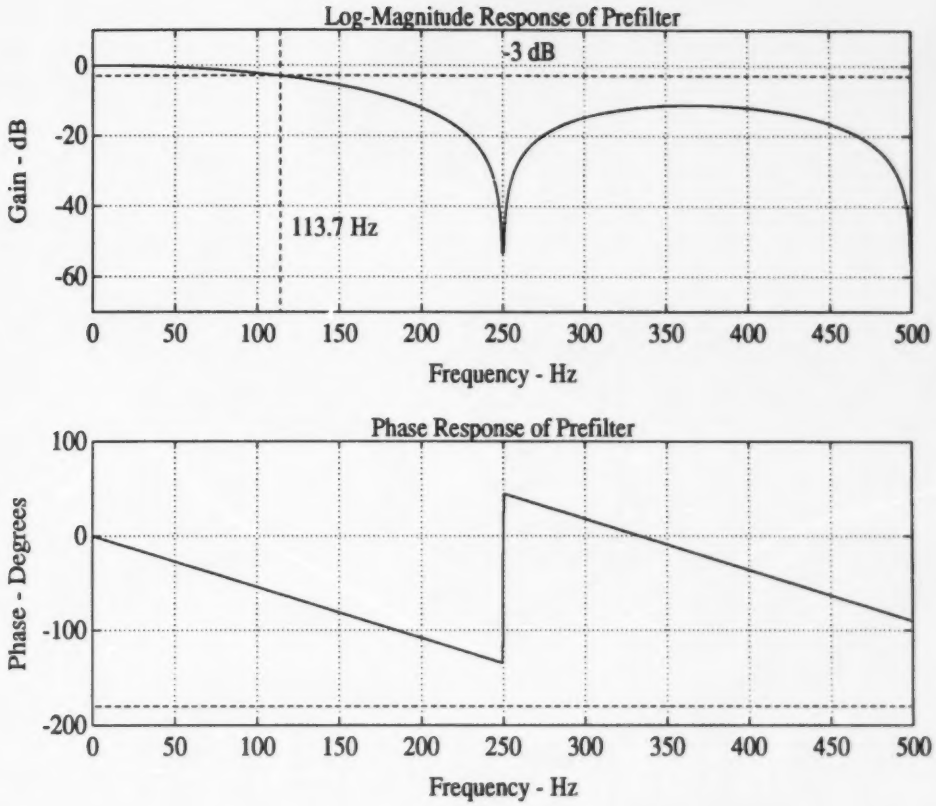


Figure 8: Frequency Response of the Prefilter

a set of smoothed joint voltages once per sample period, while Kruszewski's filter produced results once every four samples. The log-magnitude and phase responses of the new filter are shown in Figure 8. The -3 dB point on this filter occurs at about 113.7 Hz.

To achieve the four-point average, the filtering program implements the following difference equation (where $y$ is the output, and $x$ is the input voltage):

$$y(k) = \frac{x(k) + x(k-1) + x(k-2) + x(k-3)}{4} \tag{5.27}$$

The filtering software implements the division by four using shifts of the sampled data. When the voltage of an input port is sampled, it is presented in a signed 12-bit format. This is read as a 16-bit number, with the lower four bits unused. As each data sample is read, it is immediately divided by four by shifting the bits to the right two places. The current sample is then added to the time-delayed and pre-divided samples stored in a temporary buffer space and the result is placed in the output buffer. The time delay is implemented by shifting each delayed variable down to the next delay slot at the end of each sample cycle. The oldest delayed variable is discarded. A copy of the current, pre-divided, voltage sample is then placed into the $x(k-1)$ slot, and the system is prepared for the next sample period.

The filtering software also uses the sampled PUMA status lines (STOPL, SSRON, and BRKRLS [Kru90]) to drive a state machine that indicates the current *state* of the system hardware, as opposed to simply indicating which lines are high or low as was done in Kruszewski's Controller.

The prefiltering program oversamples the status lines and votes on the sampled values in order to protect against incorrect state transitions due to differing status line rise/fall times. The voltages on the status lines are sampled and converted into appropriate binary values (high or low) for each line. These binary values are then assigned to bit positions in a status code. Five such status codes are collected. If at any time during the collection of the status codes, the new status code differs from the previous one, the entire series is discarded. Once five equal status codes are collected, it is assumed the state transition of the PUMA's hardware status lines has completed, and the state machine is allowed to process the state transition. The states that the hardware may enter are defined in the header file *statemach.h* which is listed in Volume 2.

## 5.3 Configuration Files

The flexibility of the MRC is based in its use of text configuration files to define various operating parameters. The MRC utilizes four configuration files to establish associations between different control modules and their respective motion generator modules, and to set up appropriate state variable table configurations, hardware configurations, and robot home position values.

The controller's configuration mechanism uses the reconfiguration library supplied with Chimera II. These facilities allow a user to easily specify the format of the configuration file itself, as well as the data structures into which the configuration information will be written.

### 5.3.1 Controller Configuration File

The Controller Configuration file is the top-level of system configuration. This file defines the parameters of all control modules linked into the system. For a control module to be available for selection from the user interface, it must be defined in the Controller Configuration File.

Each control module is defined in the Controller Configuration File by the following eight fields:

- MODULE NAME: The ASCII name of the control module.
- MOTION GEN: The ASCII name of the motion generator module used by this control module.
- SVAR TABLE: The name of the State Variable Table Configuration File for this control module.

- DVME601 CONFIG: The name of the DVME-601 Configuration File used by this control module.
- HOME The name of the file containing robot position initialization information.
- SAMPLE PERIOD: The default sample period for the controller.
- UPDATE PERIOD: The default setpoint update period for the motion generator.
- LOGGER PERIOD: The default data logging period.

The Controller Configuration File used in this project can be found in Appendix B.2.1. This file defines four control modules; 'pid' (a PID controller), 'pidv' (a PID controller applying the derivative action to the plant output), 'pos' (a proportional controller), and 'open' (a module used for opening the control loop). The first control module in the file is the default module upon startup of the controller software.

### 5.3.2 State Variable Table Configuration File

The State Variable Table Configuration File is used to configure the controller's state variable table for a particular control module after the user has selected the desired control module. The operation of the state variable table mechanism, and the associated State Variable Table Configuration File are presented in Chapter 6 of the Chimera II Program Documentation [SSK91].

The State Variable Table Configuration File is used slightly differently in this project than presented in the manual. The 'DESCRIPTION' field in the State Variable Table Configuration file has been commandeered to provide

access permissions to the various classes of state variables that may exist in the controller. Please refer to Section 5.4 for a more detailed description of the extensions made to the state variable table mechanism for this project.

The State Variable Table Configuration File used for the PID controller in this project may be found in Appendix B.2.2.

### 5.3.3 DVME-601 Configuration File

The DVME-601 Configuration File contains information used to operate the DVME-601 in its task of sampling the PUMA potentiometer voltages and the hardware status lines. The following four fields are contained in the file:

- FILTER: Name of the Motorola S-Record file containing A/D prefiltering software.
- INIT AFB: Name of the Application Function Block file that initializes the board.
- START AFB: Name of the Application Function Block file that starts the A/D sampling operation.
- SAMPLING PERIOD: The sampling period for A/D conversions.

The DVME-601 Configuration File used in this project is presented in Appendix B.2.3. Please refer to the DVME-601 User's Manual [Dat89a] for information on Application Function Blocks. For information on the DVME-601 library and extensions implemented by this project to ease the task of writing Application Function Blocks please refer to the Software Users Guide in Volume 2 of this work.

### 5.3.4 Home Position File

The Home Position File is used to supply the information needed by the PUMA to calibrate the joint encoder counters on the VMEbus Encoder Board. The file consists of two lines, the first of which defines the home position of the robot in terms of joint angles (in degrees). The second line gives joint potentiometer voltages measured by the DVME-601 which correspond to the home position.

This file is generated by the *makehome* utility program and should not be edited by hand. Please refer to the Software User's Guide in Volume 2 for a description of the *makehome* utility.

## 5.4 RTS to NRTS Communication Library

The RTSI (Real-Time System Interface) functions are designed to provide a communications link between the Real-Time System and the User Interface. The communication link consists of passing typed and automatically sized messages between the user interface and the real-time system using the Chimera II message passing mechanism. Two message queues are established; one for messages from the user interface to the real-time system, and the other for messages in the reverse direction. Messages are sent from the user interface requesting some action or data from the real-time system, and the response is returned by the real-time system upon its completion of the action. Error checking functionality is implemented for the closely coupled FIFO queue described in the system design.

The RTSI mechanism also provides for a subset of the state variable table information held in the real-time system to be available on the host workstation for use by the user interface. Routines are available which initialize and

encapsulate the data required for user access to the controller's state variable table. The library also provides a means of easily obtaining this data from within the user interface code for error detection purposes in the user interface code.

The RTSI library provides basic typed message communication for both command and status information to be passed between the user interface and the real-time system. Commands are sent to the real-time system in the form a typed message with or without data. The types of messages available under RTSI are:

- **Commands:** such as "turn the robot on", "move the arm to this position", or "start logging these state variables".
- **Requests:** such as "return the current position of the arm", or "return the status of the real-time tasks".
- **State Variable Table Info:** such as the name, size, type, and value of a state variable.

The messages are serviced by the Robot Manager task as soon as it is able. The response to a command may be one of three possibilities;

- The requested action was successfully carried out.
- A non-critical failure occurred in the execution of the function, but the controller does not need to be shut down.
- A critical failure has occurred, and the controller should be shut down as soon as possible.

The receipt of an unexpected message is regarded as a failure in the communications queue and prompts a shutdown of the system. In the case where

a command requests that the real-time system supply some information, the receipt of the typed message containing the data is the signal that all is well.

The RTSI library provides non-real-time extensions to the state variable table system. It is not possible to attach to the real-time state variable table from the host workstation, yet certain data about the state variable table is required by the host for use within the user interface. The solution to this dilemma is to use the communication facilities supplied in Chimera II to establish a copy of relevant state variable table information on the host workstation.

The RTSI library extends the idea of the local and global copies of state variable table information to include the non-real-time system. The real-time system is used as a server which initializes and updates the state variable information on the non-real-time system (the client) when required. The real-time system also accepts requests to modify the global state variable table when the local non-real-time copy is modified.

When the user wishes to modify the value of a state variable, the local copy of the state variable is modified, and a request is sent to the real-time system to update the global copy of the state variable to the new value. Current values of the static and dynamic gains are kept on hand for reference by the user in order that the traffic on the RTSI message queue be kept to a minimum. It is also possible to declare maximum and minimum values for these gains in the state variable table configuration file for error checking purposes (see Chapter 6: Reconfigurable Systems in the Chimera II Program Documentation [SSK91]).

The information on the state variable table is gathered by the real-time system at startup (after the controller module to be used has been selected) and is sent to the user interface via the communications link. The state vari-

ables in the table are classified by the user into functional groups using the "DESCRIPTION=" field in the state variable table configuration file [SSK91]. An example of this file may be found in Appendix B.2.2. The functional groups are:

- **Controlled Variable:** This is the output of the controlled system (in this case, the joint angle). This cannot be modified directly by the user, it can only be recorded via the data logger.
- **Reference Variable:** This is the system setpoint. All access to this is through the Motion Generation task, it is not user modifiable. It is available for inclusion in data logging only.
- **Controller Output:** The output of the control algorithm. This is also not directly user modifiable by the user but may be recorded via data logging.
- **Dynamic Gain:** This is a gain that may be modified directly by the user while the controller is running. The new value will come into effect at the next cycle of the control algorithm.
- **Static Gain:** This is also a directly modifiable gain. However, the value of the variable may not be changed after the controller has been started. Setting any static gain while the controller is running will have no effect, until the controller is stopped and then restarted. The advantage of static gains over dynamic is that access to them by the control algorithm does not require repeated state variable table accesses. They are copied into the controller's local copy of the state variable table before the controller's first cycle.

- **Parameter:** This is a catch-all class for variables such as self-tuning control algorithm gains which should be included in the state variable table for logging purposes, but which should not be directly accessible to the user.

This classification of the state variables allows a certain measure of security and organization at the user interface. Often it is convenient to provide dynamic controller gains that may easily be changed by the user for tuning purposes. At the same time, there are values in the state variable table that must not be modified directly by the user, such as the controller output, or the gains generated by a self-tuning controller. However, it is still desirable to have all these variables in the same state variable table to simplify the tasks of data logging and system reconfiguration.

## 5.5 Device Libraries

To aid in the implementation of this system and to provide convenient access to the control hardware, three libraries of functions were developed for the project. These libraries utilize object-oriented programming techniques such as encapsulation to hide the details of the controller's I/O hardware from the user, and modularity to provide well defined interfaces to the hardware [Boo91].

The libraries, *libdvme601.a, libdvme628.a,* and *libenc.a*, are intended to hide the details of operating the Datel DVME-601 Analog to Digital Converter, the Datel DVME-628 Digital to Analog Converter, and the VMEbus Encoder Board respectively. The programming interfaces for the devices are identical for identical operations (such as initialization, reading, writing, and shutdown),

and differ only in the cases where the boards differ in specific capabilities (such as downloading local programs to the DVME-601).

The libraries for these devices also provide conversion from each board's native representations into equivalent floating-point values. The two's complement voltage representation of the DVME-628 and DVME-601 are converted to and from floating point voltage values for the convenience of the user by the *libdvme628.a* and *libdvme601.a* libraries respectively. The integer counts used by the VMEbus Encoder Board to denote manipulator joint angles are converted to radians by the *libenc.a* library.

These libraries conform to the suggested standards outlined in Chapter 5: System's Manual in the Chimera II Program Documentation [SSK91] for second-level device drivers. Detailed descriptions of the library functions may be found in Volume 2 in the Software Users Guide.

## 5.6 Device Drivers

The software layer responsible for communicating directly with the control system hardware and hiding details of operating the hardware from the user is the *device driver* layer. These three software modules are written using the conventions outlined in Chapter 5: System's Manual in the Chimera II Program Documentation [SSK91].

The device drivers perform board-level initialization of each of the three major pieces of control I/O hardware, the DVME-601 A/D converter, the DVME-628 D/A converter, and the VMEbus Encoder Board. The drivers also handle the actual data transfer from Chimera II software running on the RTPUs to the input or output ports on the interface cards. This function requires that the device driver software be optimized to run as fast as possible,

so as to maximize the sampling frequency of the controller.

The device drivers written for this project use the concept of 'virtual devices' [Steb] [1]. A virtual device is an object composed of a set of associated physical devices which are treated as a single unit. This enables the number of instantiations of any particular device driver to be kept at a minimum, while still allowing the flexibility of modifying which physical devices are assigned to which task.

For example, the DVME-628 D/A converter board possesses eight physical output ports. The interface hardware between the VMEbus Controller and the Unimation joint amplifiers designates six of these ports as inputs to the six robot joint amplifiers. Another DVME-628 port is used for enabling the Arm Power On switch via software control. The eighth output is not used at the present (in the future this could be the gripper control). If the device drivers were written so that there was one driver per output port, there would be seven driver instantiations required for this board alone. If the other two interface cards were also administered in the same manner, a *minimum* of 19 device drivers in total would be required for a controller. This is very costly in terms of resource usage on the RTPUs administering these device drivers. For instance, a write of six control voltages to the joint amplifier requires six separate *write()* system calls. Using the concept of virtual devices, one driver is initialized for all six output ports (they are treated as one object), and only one *write()* call is required for exactly the same operation.

The virtual devices keep the number of device drivers to a minimum and group all block accesses for common operations within one system call. The problem of mutual exclusion is solved by simply not allowing more than one

[1] Stewart refers to virtual devices as logical devices.

process to obtain access to the same physical ports through a virtual device in cases where this may cause problems. Devices which are read and not written (such as the DVME-601 A/D Converter) may be accessed by more than one process with no concerns over information corruption.

### 5.6.1 Device Driver for the DVME-601 - dadc.c

This driver is written in such a manner that all board administration tasks must be performed by the driver with minor number 0. This is done to ensure that two processes are not able to download Application Function Blocks (AFBs) at the same time, or independently modify the board's execution mode.

The driver's *dadc_read()* routine reads word size two's complement values representing voltages from whatever physical ports are assigned to that particular virtual device. These values are deposited in a buffer pointed to by the buffer argument of the *read()* system call. It is assumed the buffer is large enough to hold all values.

The *dadc_write()* routine will write either an AFB or a Motorola S-record into the appropriate place in the DVME-601's Dual Port Ram. The *write mode* must be set to the appropriate type via an *ioctl()* call before a write of either type is attempted. The write mode is reset upon the end of a write, so it must be set again before the next *write()*.

The driver's ioctl call implements five DVME-601 board level operations, in addition to the requisite assignment of physical ports to virtual devices. The operations are:

- Set the DVME-601's Executive 'on'.
- Set the DVME-601's Executive 'off'.

- Execute the last AFB loaded into the Dual Port Ram.
- Set the write mode to accept a S-record file.
- Set the write mode to accept an AFB.

The specifics on using the DVME-601 device driver may be found in the header file *dadc.h* listed in Volume 2. The details on operating the DVME-601 may be found in [Dat89a].

### 5.6.2 Device Driver for the DVME-628 - ddac.c

This is quite a simple driver, only requiring a simple output routine with no special board functions to be implemented by *ioctl()*.

The *ddac_write()* routine writes a set of two's complement voltage representations to the output ports assigned to the virtual device being accessed. The driver's ioctl call only implements the assignment of physical D/A ports to virtual devices.

More detailed information one the use and operation of this device driver may be found by examining the header file *ddac.h* listed in Volume 2. The specifics on operating the DVME-628 may be found in [Dat89b].

### 5.6.3 Device Driver for the VMEbus Encoder Board - veb.c

This driver is written in such a manner that all board administration tasks must be performed by the driver with minor number 0. This is done to ensure that two processes do not try to initialize the encoder counters at the same time.

The unique hardware of the VEB poses certain problems in developing this driver. In order to access the current joint encoder count, the values held in the counters must be latched by a write to a specific location in the VEB memory space. In order to guard against data corruption in the case of multiple devices reading the joint encoder values, the joint encoder sampling must be protected with a semaphore internal to the device driver. A local semaphore is used, since this type of Chimera II semaphore is faster than the more general remote semaphore which would work across RTPUs. Due to this, the VEB counters must not be *read* by drivers owned by different RTPUs. Regardless of this, until the remote device feature of Chimera II is fixed, it will not be possible to remotely access the VEB driver from other processors anyway.

This poses a problem where initialization of the joint encoder counter values is concerned. In this project, it is desired to have a task on a RTPU other than the one that owned the VEB driver initialize the counter values. The solution to this is to have the RTPU on which the initialization task executes own the VEB driver with minor number 0 (the administration driver). This driver is initialized as a virtual device with no physical ports. Its only function is to perform joint encoder counter initializations.

The driver's *veb_read()* routine reads word size integer values representing joint encoder counts from whatever physical joint encoder counters are assigned to a particular virtual device. These values are deposited in a buffer pointed to by the buffer argument of the *read()* system call. It is assumed the buffer is large enough to hold all values.

The driver's ioctl call implements two VEB board level operations along with the assignment of physical ports to virtual devices function. The operations are: